NASA-CR-166,072

NASA Contractor Report 166072

NASA-CR-166072
19850022394

DEVELOPMENT AND EVALUATION OF A FAULT-TOLERANT MULTIPROCESSOR (FTMP) COMPUTER
Volume II
FTMP Software

Jaynarayan H. Lala and T. Basil Smith, III

THE CHARLES STARK DRAPER LABORATORY, INC.
555 Technology Square
Cambridge, Massachusetts 02139

CONTRACT NAS1-15336
MAY 1983

FOR EARLY DOMESTIC DISSEMINATION

Because of its significant early commercial potential, this information, which has been developed under a U.S. Government program, is being disseminated within the United States in advance of general publication. This information may be duplicated and used by the recipient with the express limitation that it not be published. Release of this information to other domestic parties by the recipient shall be made subject to these limitations. Foreign release may be made only with prior NASA approval and appropriate export licenses. This legend shall be marked on any reproduction of this information in whole or in part.

Review for general release May, 1985

LIBRARY COPY

SEP 27 1983

LANGLEY RESEARCH CENTER
LIBRARY, NASA
HAMPTON, VIRGINIA

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665



NF02222

FOREWORD

This report was authored by Dr. Jaynarayan H. Lala. Dr. T. Basil Smith was the project engineer. Mr. Charles Meissner was the NASA technical monitor for the period January-December 1982, and Mr. Nicholas Murray was the technical monitor from August 1978 to December 1981. Following are some of the people who contributed to the success of this project.

Draper Laboratory

Dr. Albert Hopkins
Mr. Jack McKenna
Ms. Linda Alger
Mr. Kevin Koch
Mr. Robert Scott
Mr. Joseph Marino
Mr. David Hauger
Mr. Mario Santarelli

Collins Avionics

Mr. Ron Coffin
Mr. Charles Schulz

N85-30706#
X83-10327#

This Page Intentionally Left Blank

TABLE OF CONTENTS

Page

1. Introduction.....1

2. FTMP Software Overview.....7

2.1 Executive Software.....7
2.2 Facilities Software.....10
2.3 Acceptance Test/Diagnostic Software.....13
2.4 Applications Software.....15
2.5 Support Software.....16

3. Executive Software.....19

3.1 Basic Concepts.....19
3.2 Kernel.....33
3.2.1 Kernel Functions.....33
3.2.2 Kernel N-S Diagram.....35
3.2.3 Kernel Data Base.....39
3.3 System Restart.....45
3.3.1 Functions.....45
3.3.2 Data Base.....49
3.3.3 N-S Diagram.....53
3.4 Task Scheduling and Dispatching.....60
3.4.1 Dispatch Strategy and Implementation.....61
3.4.2 Dispatcher Data Base.....74
3.4.3 Dispatcher N-S Diagrams.....80
3.5 System Configuration Control.....95
3.5.1 SCC.....96
3.5.2 FSCC.....110
3.5.3 Fault Detection.....112
3.5.3.1 Fault Detection Methodology.....112
3.5.3.2 Fault Detection Program and Data Base.....114
3.5.4 Fault Identification.....119
3.5.4.1 Hard Failure Analysis.....120
3.5.4.1.1 Single Fault Analysis.....121
3.5.4.1.2 Double Fault Analysis.....123
3.5.4.1.3 Triple Fault Analysis.....123
3.5.4.1.4 Quad Fault Analysis.....124
3.5.4.2 Transient Fault Analysis.....124
3.5.4.3 SCC, HFA and TFA Data Base.....126
3.5.5 Spare Cycling.....127
3.5.5.1 SWAP.PROC.....129
3.5.5.2 SWAP.MEM.....130
3.5.5.3 SWAP.BUS.....130

TABLE OF CONTENTS (Cont.)

Page

3.5.6 System Reconfiguration....131
3.5.6.1 Deactivate Processor....134
3.5.6.2 Deactivate Memory....139
3.5.6.3 Deactivate Clock....139
3.5.6.4 Deactivate P Bus....142
3.5.6.5 Deactivate R Bus....142
3.5.6.6 Deactivate T Bus....147
3.5.6.7 Deactivate C Bus....147
3.5.6.8 Rotate Triad on P Bus....147
3.5.6.9 Rotate Triad on T Bus....150
3.5.6.10 Rotate Memory....150
3.5.6.11 Exchange Oscillators....151
3.5.6.12 Shadow Memory....151
3.5.6.13 Swap Processor....151
3.5.6.14 Swap Memory....152
3.5.6.15 Swap Oscillator....152
3.5.6.16 Swap P Bus....152
3.5.6.17 Swap R Bus....152
3.5.6.18 Swap T Bus....153
3.5.6.19 Swap C Bus....153
3.6 Input/Output....153
3.6.1 TTY....154
3.6.2 SENSOR.IO....157
3.7 Executive Primitives....157
3.7.1 System Bus Service Routines....158
3.7.2 Error Latch Service Routines....162
3.7.3 Timer Routines....163
3.7.4 Miscellaneous Primitives....166

4. Facilities Software....173

4.1 CTA....173
4.2 Prom Programmer....175
4.3 Fault injector software (FIS)....176
4.4 CAPS simulator....178
4.5 Miscellaneous Facilities Software Packages....179

5. Acceptance Test/Diagnostic Software....181

5.1 LRU Diagnostics....182
5.2 Opcode Diagnostics....185
5.3 Interrupt Diagostics....185
5.4 Cache Memory Diagnostics....185
5.5 Synchronization and Multiprocessing Diagnostics....186
5.6 Clock Diagnostics....186

TABLE OF CONTENTS (Cont.)

Page

6. Applications Software.....191

6.1 Console Software.....191
6.2 Flight Control Software.....204
6.2.1 APFDS.....205
6.2.2 Autoland.....211
6.3 FTMP Core Software Summary.....217

7. Support Software.....222

REFERENCE.....226

LIST OF FIGURES

Figure		Page
1-1	FTMP support environment	4
3-1	Software appearance of FTMP (Virtual Machine)	21
3-2	Structure of process state descriptor (PSD)	23
3-3	Machine state before an interrupt	26
3-4	Machine state after an interrupt	27
3-5	Machine state after resuming an interrupted process	28
3-6	Interrupt handling	29
3-7	Pend/Activate handling	31
3-8	Overall PSD structure and priority	32
3-9	Kernel N-S diagram	36
3-10	Stack initialization	38
3-11	Stack before initialization	40
3-12	Stack after initialization	40
3-13	Kernel data base - Cache PROM	42
3-14	Kernel data base - Cache RAM	43
3-15	Triad ID table	50
3-16	Memory relocation and RTC table	50
3-17	T Selects and C enables	50
3-18	P, R, and T Bus enables	52
3-19	P, R, T, and C Select table	52
3-20	Restart N-S diagram	55
3-21	FTMP workload characterization	62
3-22	Dispatcher overload performance	67
3-23	Major frame initiation	69
3-24	R4 frame termination/Resume R3	71
3-25	R4 frame initiation, interrupting R3 and R1	72
3-26	R4 control block	75
3-27	Triad tracker	75
3-28	Task data structures	76
3-29	N-S diagram of R4 dispatcher	81
3-30	N-S diagram of R3 and R1 dispatchers	84
3-31	Select task N-S diagram	89
3-32	Execute N-S diagram	91
3-33	SCC N-S diagram	97
3-34	Fault detection N-S diagram	115
3-35	ACTIVE.BUS	117
3-36	SPARE.BUS	117
3-37	BUS.MASK	117
3-38	LATCH and TR.LATCH	118
3-39	ERR.LATCH	118
3-40	LATCH.STATUS	118
3-41	Single fault analysis	122
3-42	Deactivate and Rotate Commands	132
3-43	Swap commands	135

LIST OF FIGURES (Cont.)

Page

3-44 Assign shadow command....136
3-45 Deactivate processor....137
3-46 Deactivate memory....140
3-47 Deactivate oscillator....141
3-48 Deactivate P Bus....143
3-49 Bus select codes....144
3-50 Deactivate R Bus....145
3-51 Simplex codes....146
3-52 Deactivate T bus....148
3-53 Deactivate C bus....149
3-54 Lock word structure....167
3-55 N-S diagram of lock....169
3-56 Lock word before being unlocked....170
3-57 Lock word after being unlocked....170
3-58 N-S diagram of unlock....170
6-1 System Status Display....193
6-2 LRU Bus Assignment Display....194
6-3 Failure Log Display....195
6-4 Transient Failure Log Display....196
6-5 I/O Port Status Display....197
6-6 Autopilot/Flight Director Status Display....199
6-7 CWS - Pitch Loop....206
6-8 CWS - Roll Loop....206
6-9 Lateral SAS....209
6-10 Altitude hold....209
6-11 Mach hold....210
6-12 Vertical speed hold....210
6-13 Heading hold....212
6-14 Autoland - Airspeed....213
6-15 Autoland - Glide slope....213
6-16 Autoland - Flare....214
6-17 Autoland - Lateral....216
6-18 FTMP Load Module Memory Map....218
6-19 FTMP Address Spaces....220

CHAPTER 1

INTRODUCTION

This report is Volume II of a multi-volume report on the Fault Tolerant Multiprocessor (FTMP) project sponsored by the Langley Research Center of the National Aeronautics and Space Administration under contract NAS1 - 15336. The major topic covered by this volume is the software developed under this project. A prerequisite for understanding this report is some knowledge of the FTMP architecture and its principles of operations described in Volume I. It is assumed here that the reader is familiar with the contents of Volume I.

This report is intended to be a comprehensive functional description of the FTMP software with special emphasis on the FTMP Executive and Applications Programs. The finest level of detail of the software description treated under this document is the flow-chart level, in most cases. The actual source language description of the programs and the outputs of compilers, assemblers and linkers are cited as references. The reader is urged to consult these reference documents for the instruction level details of the FTMP software.

The FTMP software can be divided into five broad categories: Executive, Diagnostics, Applications, Support, and Facilities. The

Executive is responsible for the orderly start of the FTMP, maintaining system integrity in the presence of hardware faults and timely execution of applications tasks. The Diagnostic software consists of off-line and on-line system test and self-test programs. The off-line diagnostics run in a stand-alone mode while the on-line self-test programs are intended for real-time in-flight diagnostics and are dispatched by the Executive. Applications programs constitute the core of the FTMP software from the FTMP user viewpoint. They consist of aircraft navigation, guidance, and flight control algorithms tailored for a Boeing 707 type commercial transport aircraft. All the flight control modes presently available on Lockheed L-1011 Tristar are included in this software package. The Support software provides the necessary tools to convert all these programs into a format suitable for execution by the FTMP. These tools include a cross-compiler, host-compiler, cross-assembler, and a linker. Finally, the Facilities software is a package of miscellaneous routines that are used to run the hardware facilities surrounding the FTMP.

There are three different computers, including the FTMP, on which various parts of the FTMP software are executed. These are the FTMP, a PDP-11/60, and an AMDAHL-470/V8. A hardware description of the FTMP can be found in Volume I of this report. The PDP-11/60 is a typical minicomputer that supports a multiuser RSX-11M operating system. The AMDAHL-470/V8 is a large mainframe computer that supports the MVS operating system. The PDP-11 is connected to the Amdahl computer through a TSO (Time Share Option) link. The PDP-11 is linked to the FTMP via the Test

Adapter. Figure 1-1 shows a block diagram of various hardware elements and their interconnections.

All the software that is intended to be run on the FTMP, that is, the Executive, Applications software, and Diagnostic software, is written in either the high-level language AED or the CAPS-6 assembly-level language. The AED programs are compiled on the Amdahl using the AED cross-compiler. This compiler produces object code for the CAPS-6 processor. The assembly language programs are assembled using a CAPS-6 cross-assembler on the Amdahl computer. The linker, which also runs on the Amdahl, links various object code libraries to produce a load module suitable to be loaded into the FTMP. The AED cross-compiler itself is written in AED. It is compiled using the AED host compiler which is also resident on the Amdahl.

All the facilities software is resident on the PDP-11. It is written in the PDP-11 assembly language and FORTRAN. The FTMP load modules are down-loaded from Amdahl into the PDP-11 using the TSO link that connects these two computers. The load modules are stored in PDP-11 on an RL01 disk-pack. They can be transferred from there to the FTMP system memory or cache by using a loader program which is part of the Facilities software. The load module can also be transferred into individual PROM cards. To do this, the PROM card containing 16 PROM's is inserted in the PROM programmer socket that has been attached on the PDP-11 UNIBUS. It may then be programmed using the PROM programmer routine which is also part of the facilities software.

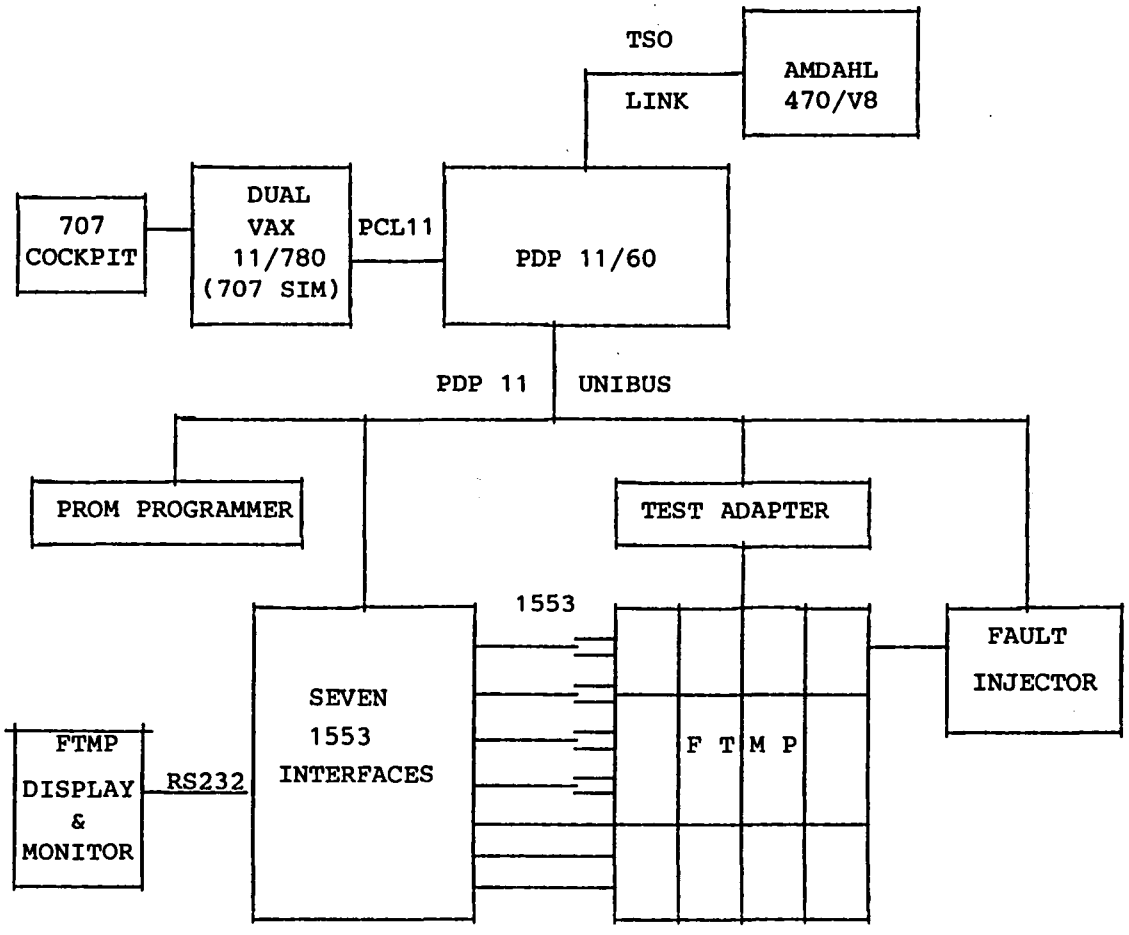


Figure 1-1. FTMP Support Environment.

It may be mentioned here that the software that runs on the FTMP, that is, the Executive, Applications programs, and Diagnostic programs, has been written following structured programming techniques. There are no 'GOTO' statements in this software.

Chapter 2 gives an overview of the FTMP software. Chapters 3, 4, 5, 6,and 7 describe the Executive, Facilities, Diagnostics, Applications, and Support software, respectively.

This Page Intentionally Left Blank

CHAPTER 2

FTMP SOFTWARE OVERVIEW

The FTMP software can be conveniently divided into five functional categories. These are described in the following five sections.

2.1 Executive Software

The Executive software is responsible for managing the hardware and software resources, for maintaining system integrity and for timely and orderly execution of applications tasks. It forms a link between the FTMP hardware and the user programs. The FTMP hardware, along with the Executive, project an image of a virtual machine to the user such that the hardware redundancy and redundancy management become transparent to the user. The user is only aware of a multiprocessing environment in which several processors execute tasks in parallel and are linked together by a single shared memory.

The part of the Executive software that is responsible for the execution of the applications tasks is called the Dispatcher. The Dispatcher is at the heart of the Executive. The remaining executive or systems tasks, that is, all systems tasks other than the Dispatcher, are treated by the Dispatcher like applications or user tasks. These other

systems tasks include a system configuration controller, system status displays and self-test programs. They are scheduled to run by the Dispatcher just the same as user tasks.

All the user and systems tasks are repetitive in nature. They are executed at one of three different iteration rates. The three rate groups, called R4, R3, and R1, presently execute at 20, 10, and 2.5 hertz, respectively. The highest frequency tasks, that is, those in the R4 rate group, are given the highest priority, while the lowest frequency tasks, that is,those in the R1 rate group, are given the lowest priority for execution. The configuration controller and display tasks are dispatched at the R1 rate.

The Executive is a timer-interrupt driven "floating" executive. Each iteration of the R4 rate group tasks is initiated by a hardware timer interrupt. The Executive can run in any processor triad. Hence the name floating. In other words, no processor triad is a master or slave triad. All triads have equal authority. Of course, only one triad is allowed to alter the Executive data bases at any given time. Access to the shared data bases, such as task queues etc., by different processor triads, is controlled by semaphores resident in the system memory. The timer interrupt initiates R4 task iteration or frame in just one triad. The other triads follow the lead triad when told to do so through IPC (Inter-Processor Communication) registers. At the completion of an R4 frame, whichever triad happens to dispatch the last R4 task becomes responsible for initiating the next R4 frame using the timer-interrupt.

The second element of the Executive is the system configuration controller. The configuration controller detects hardware faults by

analyzing information from the error latches. It is also responsible for identifying faulty units and reconfiguring the system to replace them with spares, or gracefully degrading the system if no spares are available. Faults are isolated to the LRU (Line Replaceable Unit) sub-unit level such as processor, memory, clock, I/O port, and system bus line. Weak intersections of LRU's and buses, that is, an LRU unable to transmit or receive on a particular bus line, are also identified. Faults that do not persist long enough to be isolated to one of the aforementioned sub-unit levels are handled by a transient fault analysis algorithm. Demerits are assigned to all the concerned sub-units when a transient fault is observed. The accumulated demerits are then analyzed for statistical significance to locate the source(s) of transient faults. In addition to on-demand FDIR (Fault Detection, Isolation and Recovery) the configuration controller periodically checks critical hardware elements using self-test programs. Examples of such items are voters and error latches. Spare sub-units are also constantly cycled into active state to ascertain their integrity. These include processors, system memory units, fault-tolerant clock elements, I/O ports, and system bus lines.

The third element of the Executive software is the bootstrap-/restart program. This program is responsible for bootstrapping the FTMP when the system is "cold", that is the system memory has not been initialized. In the cold start case, the system memory is initialized using an external device such as a cassette tape on a 1553 remote terminal. The bootstrap program itself is resident in the cache PROM of each processor unit. The restart program is also responsible for restoring the system to

a correct operational state after a power interruption. In this case the system memory is already loaded with all the programs, data, and the pre-power interrupt system configuration. The FTMP is brought up to this configuration and all the relevant data bases such as task queues for the dispatcher are re-initialized.

The fourth element of the Executive is the cache memory management. The cache RAM in each processor unit is only 8k words, while the total amount of data and programs in the system memory is 32k words. Therefore some method is necessary to page programs/data in and out of the cache. This is accomplished by an on-demand paging algorithm that reads the required page of data/program from the system memory into a cache page. The cache page is chosen on a round-robin basis.

The last element of the Executive is really a hardware enhancement function. The CAPS-6 processor does not have vectored interrupts. This function is now provided in the software. That is, all interrupts are vectored to their respective interrupt handling routines through the Executive software.

Chapter 3 describes the Executive software in detail.

2.2 Facilities Software

The FTMP is supported by a PDP-11/60 computer for loading programs into its system memory and to perform numerous other support functions. These other functions include test adapter commands, programming FTMP processor PROM's, fault-injector commands, and a CAPS-6 processor simulator. Software to implement these functions resides on the PDP-11 and is written in the PDP-11 assembly language or in FORTRAN IV Plus.

The first of these facilities software packages is the test adapter interface program, called the CTA (Collins Test Adapter program). The test adapter is the keyboard interface to an LRU of the FTMP. It can be connected to an LRU through the processor region transfer bus. At the same time, the test adapter can be connected to the PDP-11 on the Unibus. The CT program provides on any PDP-11 terminal all the functions that are available on the test adapter keyboard such as halt, run, reset, etc.

Additionally, all the FTMP control registers can be accessed from CTA using mnemonics. To access the system memory and other registers which are on the FTMP system bus and not directly available on the processor region transfer bus, it is necessary first to load a cooperator program (COOP) in the master LRU cache with the help of CTA. Having done this, CTA can access all the system bus devices through the COOP program. For example, the FTMP system memory can then be loaded from a load file resident on a PDP-11 dispack. Other system bus devices such as the Real Time Clock, Error Latches, etc. can then also be accessed from a PDP-11 terminal using CTA.

The PROM programmer package provides all the necessary commands at a PDP-11 terminal to program eight "2716" PROM's contained on a single FTMP cache card. The hardware to perform this function is attached to the PDP-11 Unibus. The data for programming the PROM's is obtained from the standard FTMP load module resident on the PDP-11 disk. This load module, however, can be edited and modified prior to being used to program the PROM's.

The fault-injector software (FIS) package provides commands at a PDP-11 terminal to perform all the functions necessary to inject faults into the FTMP and observe the results. The fault-injector hardware is attached to the PDP-11 Unibus and consists of six cards, each of which can be interfaced to eight pins of an IC package on the FTMP. Signals, simulating hardware faults, can be injected on any combination of these 48 pins from the PDP-11. The FIS program can be used to define the subject IC characteristics (number of pins etc.), mapping of these pins into the fault-injector pins, description of faults to be simulated on each pin and actually inserting the faults. It also keeps record of the result of each fault injection by collecting data sent by the FTMP to the PDP-11 via a 1553 interface. This data includes identification of the faulty unit, time taken to identify the fault, etc. FIS, once initialized for a given fault set, can perform the fault injection repeatedly for any desired number of times without manual intervention. The 1553 link between the PDP-11 and the FTMP is used by FIS to communicate to the FTMP configuration controller and make sure that the previous fault has been identified and the system has been reconfigured to receive and identify the next fault.

The last part of the facilities software package is the CAPS-6 processor simulator. The simulator provides a uniprocessor environment with some of the multiprocessor functions such as inter-processor communication registers and control registers on all 12 LRU's. This simulator was used to debug parts of the FTMP Executive software, mainly the Kernel, Dispatcher, and the Restart program, prior to the delivery of the

FTMP hardware. Specifically, this package simulates the CAPS-6 instruction set, some control registers such as CPU control registers, IPC (InterProcessor Communication) registers, Bus Guardian Unit registers, Error Latches etc., 12K processor cache memory an 48k system memory. Additionally, it also simulates the Real-Time Clock and timer, IPC, and stack overflow interrupts.

Chapter 4 describes the facilities software in detail.

2.3 Acceptance Test/ Diagnostic Software

A set of acceptance test programs was written at Collins Avionics and at CSDL to verify FTMP hardware operation prior to the delivery of the hardware to CSDL. These programs have since been modified and expanded at the CSDL to extend their scope to more areas of the FTM hardware.

Basically, each of these diagnostic programs runs in stand alone mode without any support from the FTMP Executive. That is, they are used to exercise the computer in a ground support environment. Typically, the LRU to be tested is made the master LRU by inserting the master plug into it. The master LRU is then connected to PDP-11 via the transfer bus and the test adapter. The appropriate dignostic program is then loaded into the cache RAM of the master LRU from PDP-11 using the CTA program (see Section 2.2). The results of the test (pass/fail) appear in the test adapter display registers. If the test is a failure, a reason code indicating which part of the test the LRU failed is also shown. Parts of the LRU which are tested in this manner include the processor instruction set, interrupts, cache RAM, system memory contained in the LRU, I/O port

registers, system control unit registers, and interfaces between the LRU and the system bus. To ascertain complete system integrity, these diagnostic tests must be performed on all the LRU's, one unit at a time.

There are other architectural features of the FTMP which can not be tested for correctness of operation by exercising LRU's individually. Examples of these include synchronous operation of three processors, bus contention and arbitration between several processors, phase-locked operation between various clocks, etc.

To test these aspects of the FTMP operation, a number of LRU's must be started up. This is accomplished by loading system memory with the appropriate diagnostic program via the master LRU. Each processor required to participate in the test is then enabled on all the buses and one of its control registers initialized such that upon being reset the processor loads its cache from the system memory and waits for further instructions in its control registers. A number of processors can thus be loaded with identical programs without moving the test adapter cable (the transfer bus from one LRU to the next. Once all the LRU's participating in the test have been loaded with appropriate programs the master LRU hogs the system bus, issues commands to the candidate LRU's by writing into their respective control registers and then releases the system bus. The master LRU then observes the results by continually monitoring the system memory locations containing the results.

Chapter 5 describes the diagnostic programs in detail.

2.4 Applications Software

Since the FTMP is intended as a central processing complex for a commercial transport aircraft, the applications programs have been written to fulfill that function. The applications programs on the FTMP perform almost all the flight control functions found in a modern commercial transport aircraft such as the Lockheed L-1011 Tristar. These include guidance and navigation, air data computation, basic stability augmentation, and various modes of the autopilot/ flight-director system for all the flight regimes, that is, take-off, climb, cruise, descent, approach, and landing.

The basic stability augmentation system (SAS) provides yaw damping and turn coordination during cruise, roll damping when the autopilot is not engaged and yaw damping, runway alignment and roll-out during autoland operations. The autopilot/flight director system provides a basic control wheel steering (CWS) mode of operation in which the autopilot holds a constant pitch and roll attitude and the pilot's pitch and roll stick inputs are interpreted as rate commands to change the aircraft attitude. For the climb and descent phases of flight, three different modes of operation are available. These are the vertical speed hold, the indicated airspeed (IAS) hold and Mach hold. During the cruise phase the Mach hold or altitude hold modes of the autopilot may be engaged. Additionally, a heading hold and VOR tracking can be used during any of these flight regimes for directional guidance and control. Localizer/VOR tracking is provided for the approach phase and localizer/glide slope tracking (Instrument Landing System or ILS) are used for autoland operations.

Speed is controlled by modulating thrust if the autothrottle mode is used. Two autothrottle modes of operation are available: constant airspeed or constant angle of attack.

The control laws have been chosen for a Boeing 707 type aircraft. Dynamics of such a type of aircraft are simulated on the dual VAX-11/780 computers of the Draper simulation facility. This simulation facility also includes a fully instrumented flight deck of a 707 driven by the VAX computers. Figure 1-1 shows the communication links between the FTMP and the 707 cockpit. The aircraft and engine instruments and the autopilot mode select switches are sent from the VAX computers to the PDP-11 over a high speed data link (DEC PCL11). From the PDP-11 the data is transmitted over MIL-STD 1553 avionics data bus to the FTMP. To send actuator commands back from the FTMP to the aircraft simulation, the reverse route is used. There are six 1553 bus interfaces between the PDP-11 and the FTMP. Four of these are connected to two I/O ports each in the FTMP and the remaining two are tied to one port each. Thus any one of the ten I/O ports on the FTMP may bused to communicate with the 707 simulator.

The applications software also includes programs to display FTMP status information on a console and accept operator commands from the console.

Chapter 6 describes the applications programs in detail.

2.5 Support Software

All the programs that run on the FTMP are written either in a high level language called AED or the CAPS assembly language. There are no

facilities on the FTMP itself to compile or assemble these source programs. These programs must be converted into the FTMP machine language on one of the mainframe computers such as IBM 360/370, UNIVAC 1108, or Amdahl 470 for which the support software is available. At CSDL an AED cross-compiler and a CAPS cross-assembler have been installed on an Amdahl 470/V8 which is the CSDL central computing facility.

The AED cross-compiler compiles and assembles AED source programs to produce relocatable object code modules. The CAPS cross-assembler assembles CAPS assembly language programs into relocatable object code modules. A cross-linker, also resident on the Amdahl, links these libraries of relocatable object code modules into an absolute load module. Each of these functions can be performed either in a background batch mode or on-line using the interactive TSO (Time Share Option) facility available on the Amdahl. There is a direct TSO link between the Amdahl and the PDP-11 (see Figure 1-1). The FTMP programs can be interactively created, edited, compiled, assembled and linked directly from any terminal that is attached to the PDP-11 computer. Having created an absolute load module on the Amdahl, either in the batch mode or on TSO, the load module can be transmitted over the TSO link to the PDP-11 where it is stored on an RL01 disk pack. The load module produced by the cross-linker is in hexadecimal format. Prior to being transmitted to the PDP-11 it is converted into an ASCII character format. The load module contains a complete core image or memory image of all the programs that are to reside in the FTMP. This, for example, includes the Executive, all the applications programs and all the data bases. If one of these programs or data bases needs to be

altered then a new load module containing the new core image must be produced and down-loaded into the PDP-11. Each record of the load module contains an identification tag indicating its eventual destination in the FTMP such as the cache PROM, the cache RAM or the system memory. The loader which is resident on the PDP-11 checks the identity of each record and loads it into the appropriate section of the FTMP memory. This same load module is also used to program the FTMP PROM's (see Section 2.2).

In addition to the AED cross-compiler, an AED host compiler is also available on the Amdahl. The host compiler compiles AED source programs and produces an object code that can be directly executed on the Amdahl. Since the AED cross-compiler itself is written in AED, the host compiler is used to compile the cross-compiler. It may also be used to debug FTMP programs on the Amdahl provided certain data type declarations in these programs are modified to make them compatible with the Amdahl data types.

Details of the support software are described in Chapter 7.

CHAPTER 3

EXECUTIVE SOFTWARE

There are certain prerequisite concepts which are central to a thorough understanding of the FTMP software. Examples of these concepts are the software appearance of the machine, definition of a "process" in the context of the FTMP, transfer of control between processes, and the handling and use of interrupts in the multiprocessor. Section 3.1 defines and elaborates these concepts. Section 3.2 details part of the Executive, called Kernel, which implements some of the basic primitives defined in these concepts. The rest of the executive software can be divided into five major functional modules: System Restart, Task Dispatch, Configuration Control, Input/Output, and Executive Primitives. The functions, program implementation, and data structures for each of these five elements ofthe Executive are described in the succeeding five sections.

3.1 Basic Concepts

The software appearance of the fault tolerant multiprocessor is much simpler compared to the actual hardware. This is due to the FTMP architecture which hides most of the hardware redundancy from the

programmer. Figure 3-1 shows the FTMP from a software viewpoint. The basic architecture of the virtual FTMP is that of a multiprocessor with three processors sharing a single system memory which can be accessed by only one processor at a time over the system bus. Also available over the system bus are a set of 11 I/O ports, each of which is connected to a MIL-STD-1553 avionics data bus. A 32-bit real-time clock and 48 error latches, which record any errors on the system bus, can be read over the system bus just like any other memory location. The bus contention between the processors as well as the hardware redundancy underlying various elements is transparent to the programmer. The applications programmer, for example, does not need to know the fact that each processor really is a set of three processors or that the 32k system memory is really two 16k modules each of which is also triplicated, in order, for example, to be able to write the flight control software. This simplified software appearance of the multiprocessor extends to some of the executive software also. The task dispatcher, for example, need not concern itself with the processor, memory, or the system bus redundancy, while the system restart and most of the configuration controller do. The hardware details of the FTMP, beyond what is shown in Figure 3-1 as the virtual FTMP, will be described where necessary in this chapter.

While the overall architecture of the FTMP is similar to that of a conventional multiprocessor, each processor of this multiprocessor is multiprogrammed to work on several "processes" simultaneously. Although strictly speaking, a processor can be working only on one process at any given instant, several processes can be in different stages of progress

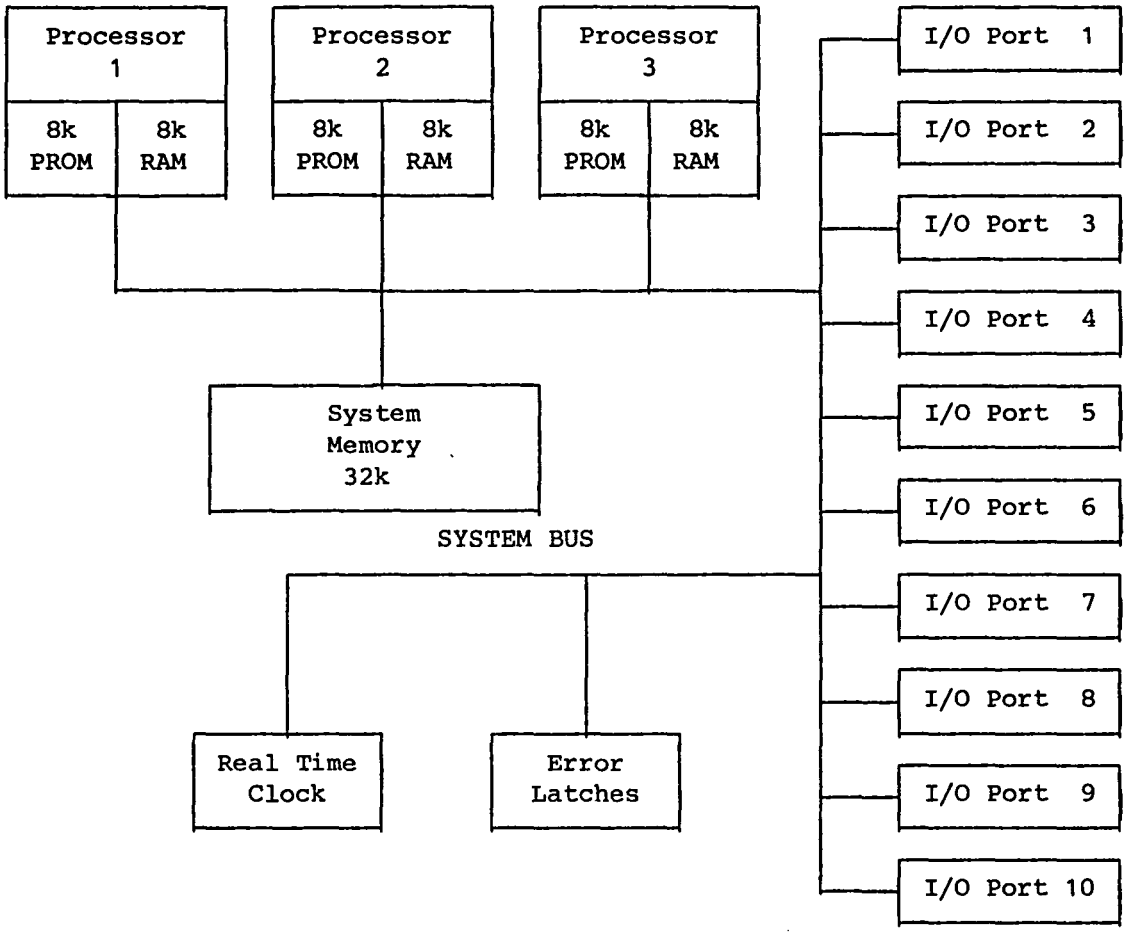


Figure 3-1. Software appearance of FTMP (virtual machine).

at the same time. The processor can suspend work on one process and switch its attention to another process in an orderly manner. To understand this capability of the Executive to multiprogram each processor, it is necessary first to define a "process" in the present context. A "process" is simply a "state of the machine," that is, the hardware. The concept of process is sometimes confused with a program. But a program is not a state of the machine, it is a set of precise instructions to be followed by the machine.

Associated with each process is an 8-word data base called the Process State Descriptor or the PSD. A PSD is a snapshot of a program at a given instant in time. Such a snapshot contains enough information to restore the machine back to its original state when it suspends that process. A PSD, thus, is a convenient way of switching context and working simultaneously on several processes. Figure 3-2 shows the eight components of the PSD. The first two words of the PSD describe the boundaries of the stack space allocated this process. The third word is the SPCR (Syllable Program Counter Register), that is, the next instruction of the process. The fourth word is the LENV (Local Environment) Register or the pointer to the memory area containing all the local variables for the process. The next word indicates whether the processor is in the privileged mode (PMR=1) or the user mode (PMR=0) when executing this process. Certain instructions, including the system bus access, can be executed by the process only when it is in the privileged mode. All of the executive software runs in the privileged mode. The next word indicates whether the mapper is turned off (MAPPER=0) or on

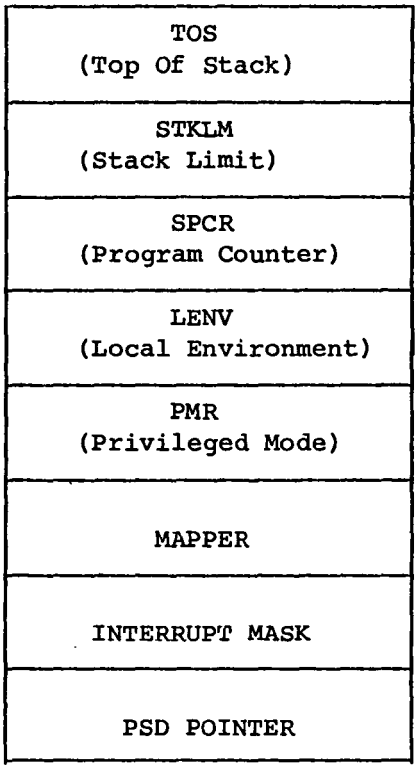


Figure 3-2. Structure of Process State Descriptor (PSD).

(MAPPER=1), that is, whether the cache memory is to be accessed directly (program addresses are real) or by mapping the program address through the memory mapper (program addresses are virtual). All of the executive software, with the exception of the Restart Program and the Page Fault Handler, runs in the mapped mode. The seventh word is the interrupt mask and indicates which of the eight hardware interrupts are to be allowed during the execution of this process. The last word points to the PSD of the process that should be resumed when the present process terminates. A process terminates when the processor executes the HALT instruction.

Normally when a process is active, that is, the processor is executing that process, all the information about its state is actually distributed in the stack and control registers. At this time, the PSD associated with this process, which is pointed to by APSD (Active PSD), is empty or has no useful information in it except for the pointer to the next PSD. When this process terminates, its PSD is filled in with the appropriate information as shown at the time by various registers. These registers are then restored with the contents of the new PSD which is pointed to by the last word of the old PSD. These two functions, viz. saving the state of the machine in the old PSD and restoring the state of the machine with the new PSD, would normally be performed in the hardware. However, since the CAPS-6 processor does not have this capability, the executive software provides this hardware enhancement function. A program written in the CAPS assembly language, FTMP.ASM(KERNEL), is entered when the processor terminates a process by executing the HALT instruction. The Kernel saves the state of the machine in the old PSD

and loads the registers with the new PSD and then transfers control to the new PSD by executing the INTRTN (interrupt return) instruction.

The Kernel actually performs several other hardware enhancement functions. One of these is "interrupt vectoring". When an interrupt occurs that is not masked in the active process, the active process is suspended and control transfers to the Kernel. The Kernel saves the process state in the PSD, obtains the pointer to the new PSD associated with the interrupt handler and loads the machine with the new PSD. The pointers to interrupt handling processes are stored in a table called the INT.TABLE. The interrupted PSD is chained behind the interrupt PSD by saving the pointer to the interrupted PSD in the interrupt PSD. The APSD is changed to point to the interrupt PSD. The Kernel then executes INTRTN instruction to transfer control to the interrupt process. Figures 3-3 and 3-4 show the state of the two processes involved in this sequence of events before and after the interrupt, respectively. When the interrupt process terminates by executing HALT, control transfers to the Kernel which restores the previously interrupted process from its PSD and the processor resumes execution of that process at the precise point where it was interrupted. This is shown in Figure 3-5. The HALT instruction, which is used to terminate a process, itself causes an interrupt, thereby transferring control to the Kernel. Since HALT in this machine is used in effect to terminate one process and resume another one, it has been synonymized with RESUME in the programs.

Figure 3-6 summarizes how an interrupt is handled in the FTMP. Prior to the interrupt process, Z is the active process. The interrupt

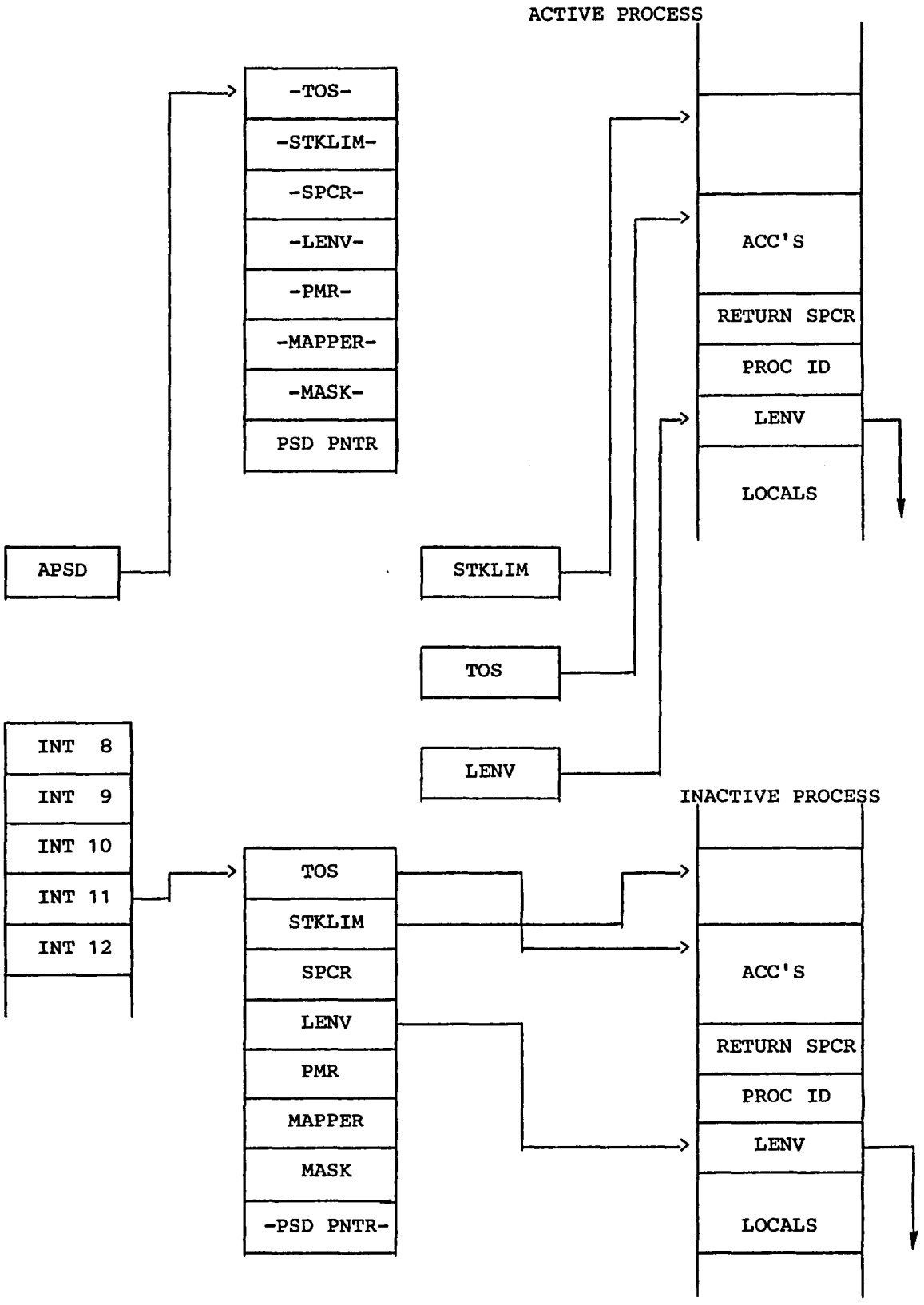


Figure 3-3. Machine state before an interrupt.

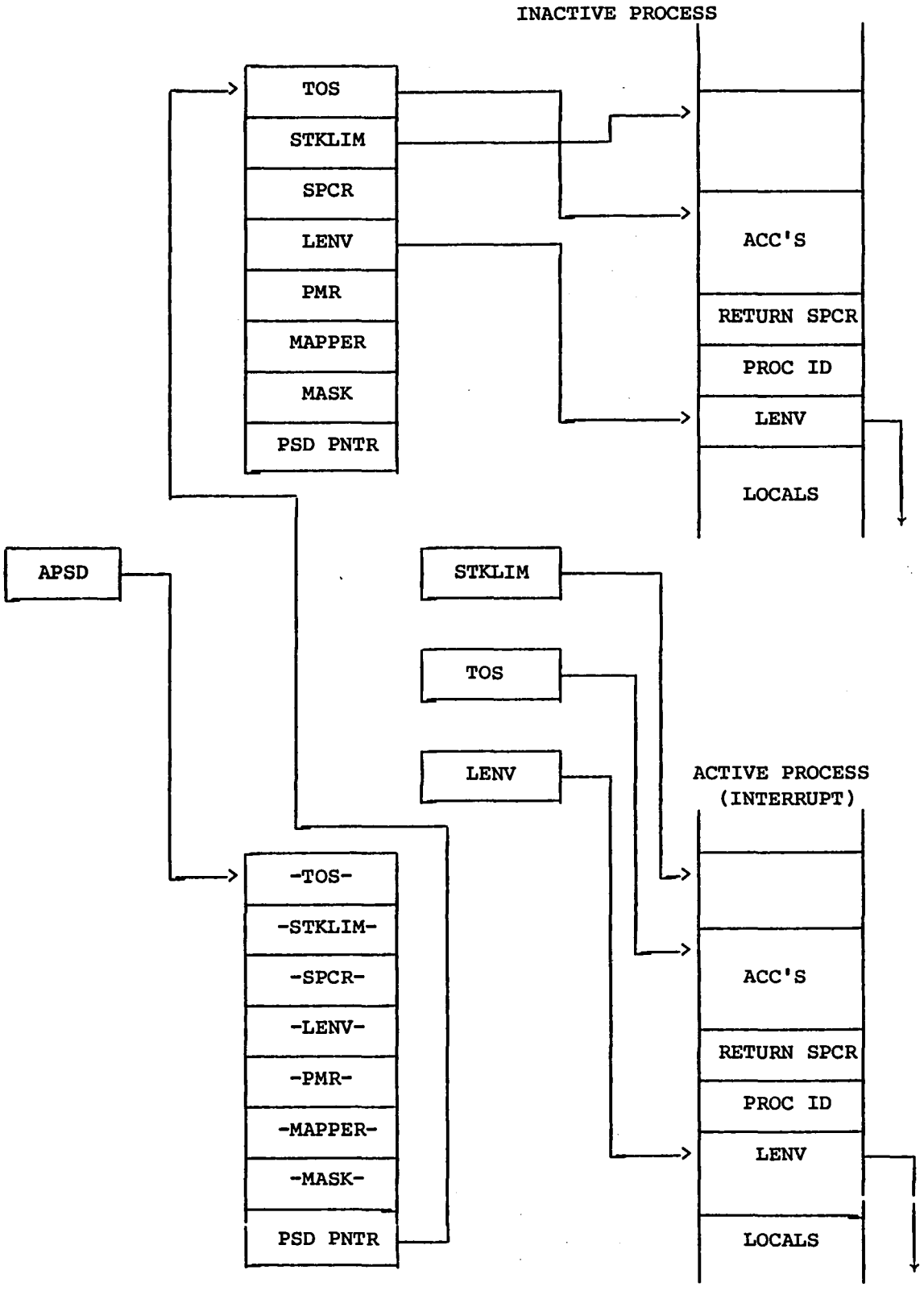


Figure 3-4. Machine state after an interrupt.

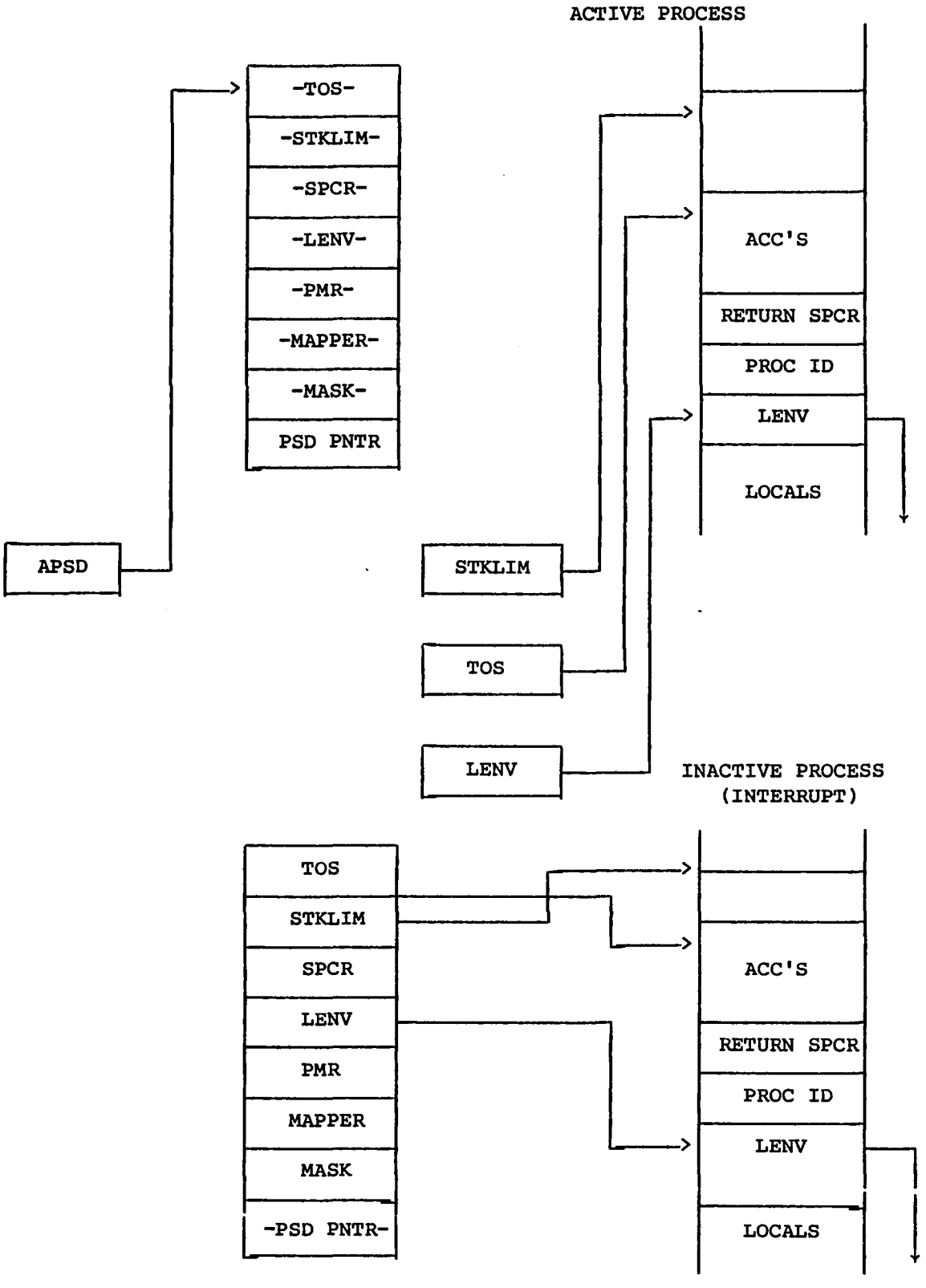


Figure 3-5. Machine state after resuming interrupted process.

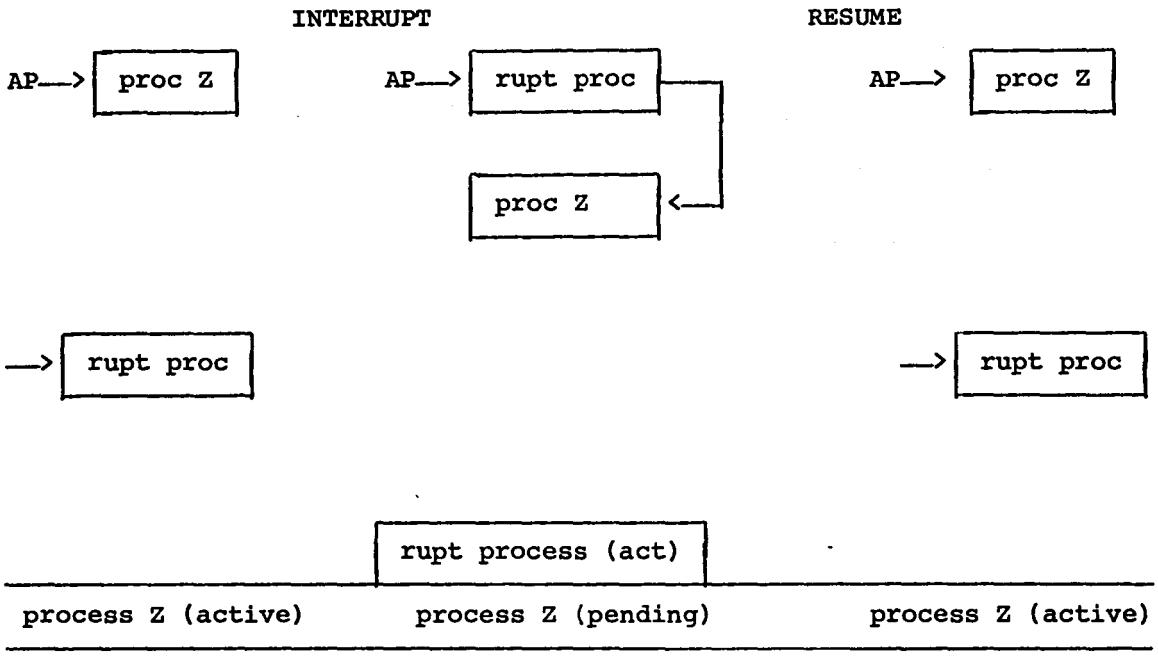
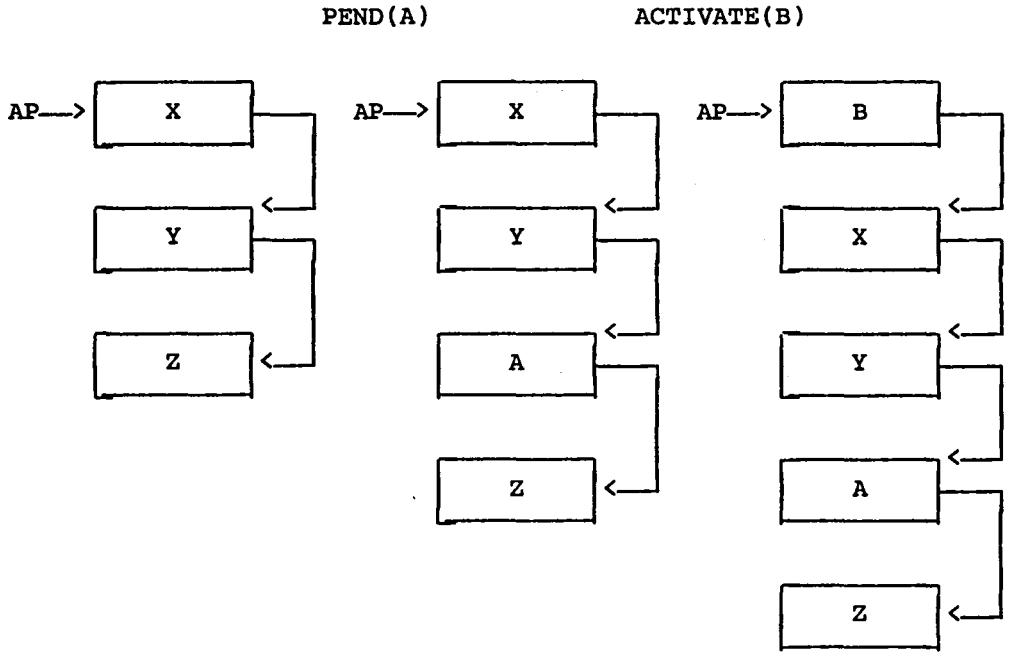


Figure 3-6. Interrupt handling.

process is inactive at this time. When the interrupt occurs, the interrupt process is made active and process Z is "pended" behind it. When the interrupt process does a "resume", the interrupt process is made inactive and the process Z is "resumed," i.e., activated.

So far, two ways have been described of transferring control from one process to another, viz. through resume or an interrupt. There are two other ways of altering the order in which processes will be executed. These are called Pend and Activate. Figure 3-7 illustrates the use of these two primitives. As seen in this figure, initially X is the active process and Y and Z are chained behind it. If process X decides to pend process A, it can do so (provided X is a privileged process) by calling the Pend routine with a pointer to the APSD as the argument. Process A is pended where its built-in priority dictates. Assuming A has higher priority than Z but lower than Y, it will be chained between Y and Z. A discussion of the PSD priority follows shortly. The Activate primitive is used to transfer control to another process as illustrated in Figure 3-7. If process X wishes to activate process B, it does so by executing the same HALT instruction which is used to resume another process but with a non-zero argument, the argument in fact being the PSD pointer to process B. Activate function is handled by the Kernel while the Pend function is performed by a routine FTMP.ASM(PEND).

There is a process associated with each of the 11 interrupts that have been defined in the CAPS-6 processor. In addition, there are eight other processes as shown in Figure 3-8. The process at the top, the overflow interrupt handler, has the highest priority and the process at



		B (active)
	X (active)	X (pending)
X (active)	Y (pending)	
Y (pending)	A (pending)	
Z (pending)		

Figure 3-7. Pend/Activate handling.

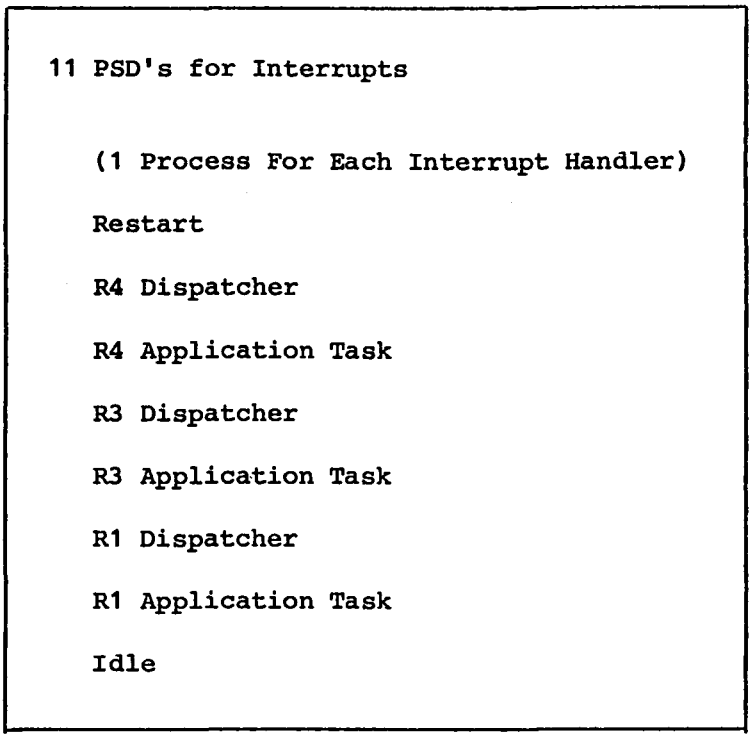


Figure 3-8. Overall PSD structure and priority.

the bottom, the idle process, has the lowest priority. This priority structure is used to pend a given process in its appropriate place in the PSD chain. The usefulness of this will be evident in the next section. As can be seen from Figure 3-8, since the idle process is at the bottom of the PSD chain, its PSD pointer is null, that is, idle PSD does not point to any other PSD. The idle process therefore should never terminate (by doing resume). Indeed, that is the case. As the name implies, a processor falls into this process when it has no other task to perform and it stays in idle until interrupted to execute another process.

Section 3.2 describes the Kernel program which implements primitives described above.

3.2 Kernel

The following three subsections describe the functions, flowchart, and data bases of Kernel.

3.2.1 Kernel Functions

This program resides in the cache PROM of each processor. When the power is first applied to the processor, a POC (Power On Clear) interrupt is generated. The microcode executes the POC reset sequence and transfers control to the program whose address appears in location 0 of the PROM. Since the Kernel address is stored at this location, the Kernel is the first program to start up after a power interruption or after the processor is reset.

Depending upon the contents of the CPU control register 3 (CPU R3), the Kernel takes one of three different courses of action. If the register R3 contains "F", a normal FTMP start is made. A normal FTMP start means transferring control to the Restart process. To do this, the Restart PSD in the cache RAM is initialized from a copy in the PROM, the stack, mapper, PMR, and other registers are initialized using information from the Restart PSD and control is passed to Restart by executing the INTRTN instruction.

If the register R3 is equal to 1, then system memory ("2005" to "2007" and "2010" to "3FDF") is read into cache RAM ("2005" to "2007" and "2010" to "3FDF"). Locations 2005 to 2007 contain the TOS, STKLIM and SPCR of the alternate program which is then started up. This start-up mode is useful when the diagnostics programs or any other programs are to be run on a single LRU in a stand alone mode without support of the FTMP Executive.

The third course of action is taken if R3 is equal to 2. This case is similar to the second case in that a program is copied from system memory into cache. However, having done this the processor waits in a loop until Register R3 is cleared. This start-up mode is used for those diagnostic programs which require several LRU's to start up simultaneously.

The next subsection describes the Kernel program with the help of an N-S (Nassi-Shneiderman) [1] diagram or a flow-chart.

3.2.2 Kernel N-S Diagram

The source code for this assembly language program is contained in FTMP.ASM(KERNEL) and the assembler output is in FTMP.LIST(KERNEL). Figure 3-9 shows a flow-chart of the Kernel. There are two entry points in this program. The first entry point is at the top of the flow-chart. The program is entered here in response to a power interrupt or if the processor is reset. If it is a normal system start, control is transferred to Restart by executing INTRTN. Executing this instruction changes the processor state from "interrupt mode" to "normal mode." In the interrupt mode no interrupts are accepted while in the normal mode all unmasked interrupts are accepted. When such an unmasked interrupt occurs in the normal mode, the active process is suspended and control is transferred back to Kernel at the instruction immediately following INTRTN. This is the second entry point of this program. This is also the only entry point used once the processor has started up. That is, the processor returns here in response to any interrupt. Upon entry, the interrupt number is in the top of Kernel stack and the next word in the stack points to the interrupted PSD. The program checks to see if the interrupt was done to Resume or Activate, in which case the interrupt number would be 16 (HALT instruction generates interrupt 16). If it is HALT, its argument is checked to distinguish between Resume and Activate. Resume's argument is zero while Activate's argument points to the PSD of the process to be activated. The HALT argument is in the TOS of the interrupted PSD. For interrupts other than HALT, the pointer to the new PSD is obtained by indexing interrupt number to the 16-word INT.TABLE

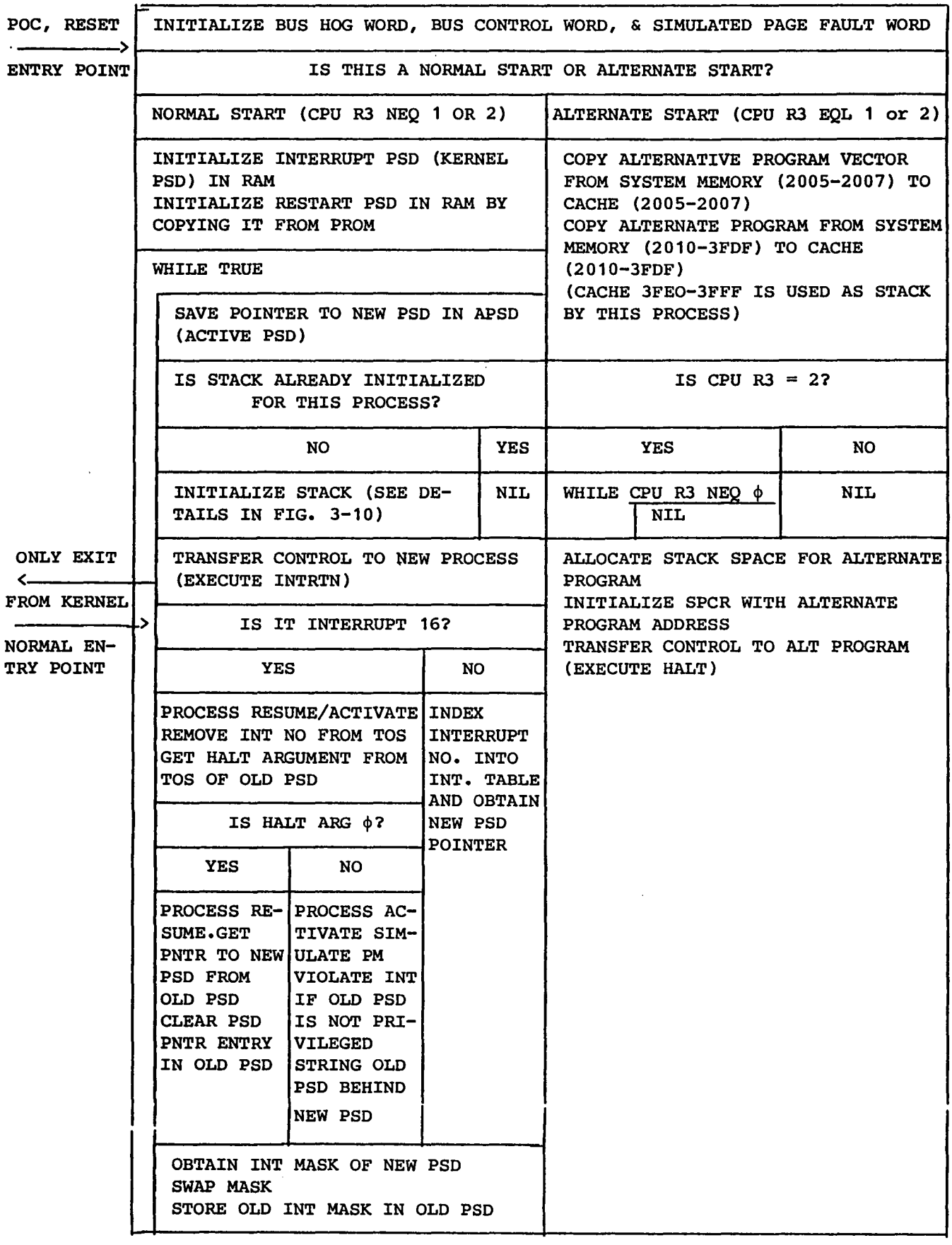


Figure 3-9. Kernel N-S diagram.

(Interrupt Table) which resides in the cache PROM of each processor. In the CAPS-6 processor, interrupts are numbered from 8 to 20. Of these, 8 and 9 are unassigned. The first entry in INT.TABLE corresponds to interrupt 8, the second to 9 and so on.

Once the pointer to the new PSD has been obtained, the machine state is saved in the old PSD, the new PSD information is loaded into the registers and various PSD's are re-linked as illustrated in Figures 3-6 and 3-7. The order in which they are chained together is, of course, dictated by which function is being performed, that is, Resume, Activate or interrupt routing. The program exits in all of these cases after saving the pointer to the new PSD in APSD, initializing the stack for the new process if necessary and executing the INTRTN instruction immediately above the normal entry point. The Kernel is then positioned to start over at the correct instruction in response to the next interrupt.

When a program is started the very first time, its stack is not initialized. This is ascertained by checking the LENV entry in the PSD. This pointer is null if the stack has not been initialized. To initialize the stack, the following steps are taken (Figure 3-10 shows the N-S diagram of this program).

(1) Obtain stack location (TOS) for this process from the PSD.

(2) Obtain the number of words to be allocated in the stack from the header word of the program. The SPCR entry in the PSD points to this word. The low order byte of the header word is the number of local and temporary words required by the program on the stack. The SPCR has the real address of the

<table>
<tr><td colspan="3">MAPPER FOR THIS PROCESS ON?</td></tr>
<tr><td colspan="2">YES</td><td>NO</td></tr>
<tr><td colspan="2">OBTAIN VIRTUAL WORD ADDRESS OF THE HEADER FROM SPCR ENTRY IN PSD.COMPUTE VIRTUAL PAGE ADDRESS</td><td rowspan="5">NIL</td></tr>
<tr><td colspan="2">IS THIS PAGE PRESENT IN CACHE?</td></tr>
<tr><td>YES</td><td>NO</td></tr>
<tr><td>NIL</td><td>WRITE THIS PAGE NO IN SFWORD
SIMULATE PAGE FAULT INTERRUPT</td></tr>
<tr><td colspan="2">TRANSLATE VIRTUAL ADDRESS OF HEADER INTO REAL ADDRESS THROUGH MAPPER</td></tr>
<tr><td colspan="3">OBTAIN NUMBER OF LOCALS, J, FROM THE HEADER WORD
ALLOCATE J WORDS ON STACK BY DECREMENTING TOS BY J AND SETTING LENV TO THIS VALUE
ALLOCATE 3 MORE WORDS FOR STACK MARK
INITIALIZE STACK MARK (SPCR, PROC ID, LENV)
(SPCR & LENV ENTRIES IN STACK MARK ARE 0, PROC ID IS THE HEADER WORD)
INCREMENT SPCR PAST HEADER</td></tr>
</table>

Figure 3-10. Stack Initialization.

header if the process runs unmapped (mapper word = 0). This address, however, is virtual if the mapper is turned on. For the latter case, the virtual SPCR is translated into the real address by mapping it through the memory mapper, provided the required page is present in cache. If not, a page fault interrupt is simulated to read the page from the system memory into the cache.

(3) The required number of words, as determined in step (2), are allocated in the stack by moving TOS. LENV is also initialized.

(4) An additional three words are allocated in the stack and a frame mark consisting of SPCR, PROC ID and LENV are written in this space. (See Figure 3-9).

(5) The SPCR is incremented past the header word.

Figures 3-11 and 3-12 show the PSD and stack before and after stack initialization.

At this point, it is appropriate to describe the data bases that are either used directly by the Kernel or defined by the Kernel indirectly. Some of these, such as INT.TABLE and Restart PSD, have already been referred to in the above discussion. All the data bases that are defined in the program FTMP.ASM(KERNEL), collectively referred to as the Kernel data bases, are described below.

3.2.3 Kernel Data Base

The Kernel data base can be conveniently divided into two parts. The first part is stored in the first page of cache PROM (Adr 0000 to

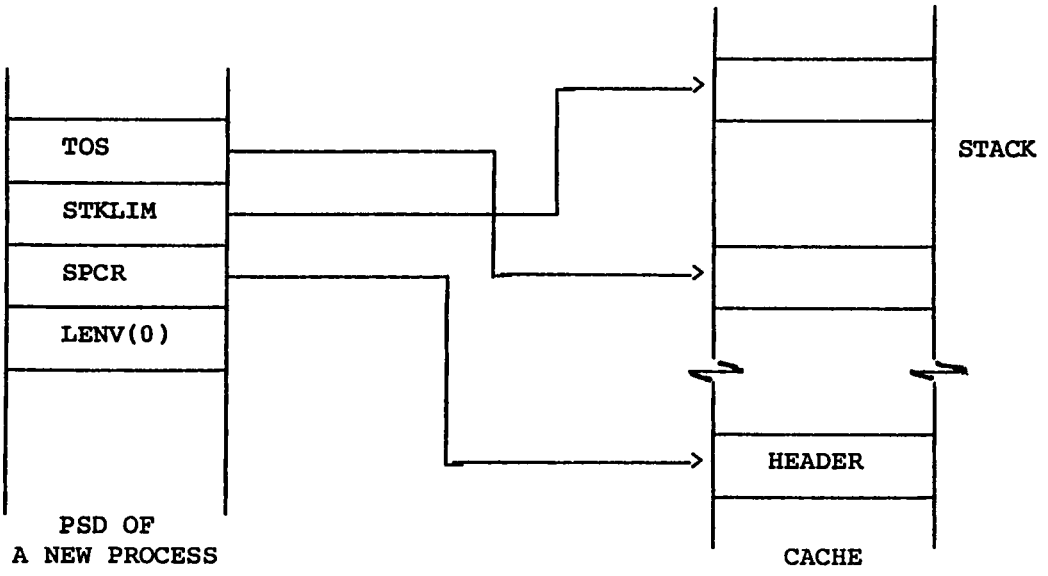


Figure 3-11. Stack before initialization.

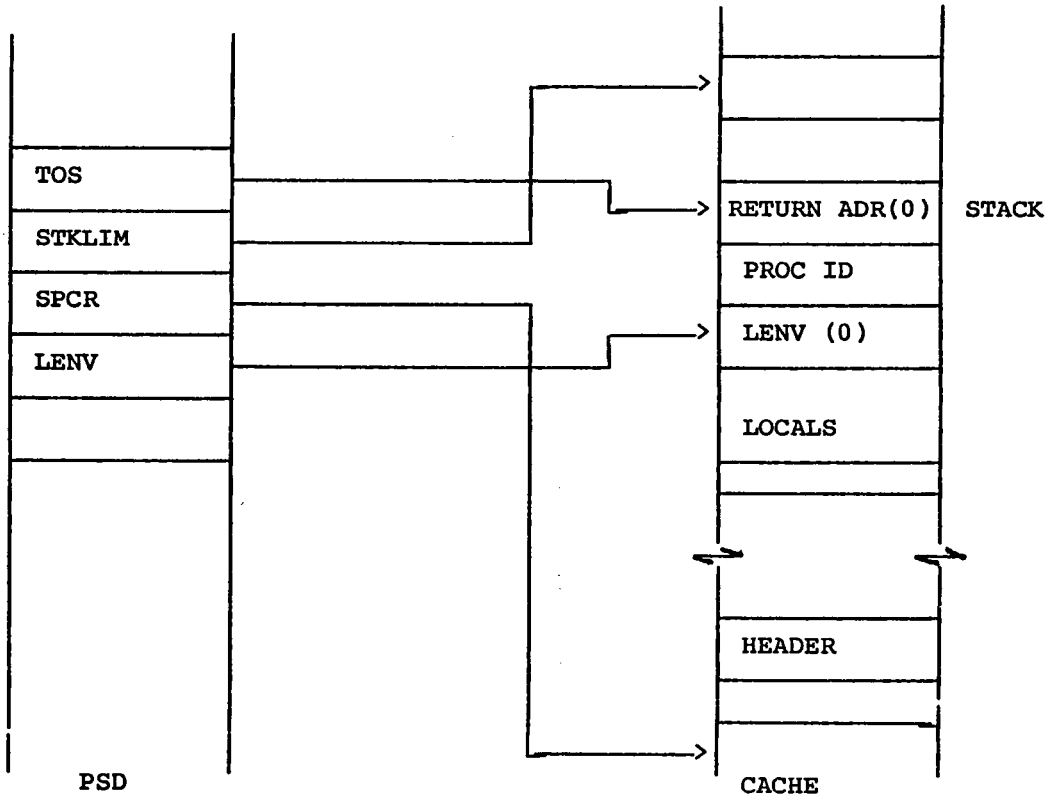


Figure 3-12. Stack after initialization.

009F). The second part is in the first four pages of cache RAM (Adr 2000 to 23D8).

Figure 3-13 shows the contents of page 1 of PROM. Locations 0, 1, and 2 in PROM define the SPCR, TOS and STKLIM of Kernel. Location 3 is the pointer to the interrupt or Kernel PSD. Location 4 points to a table of privileged mode routines (PMTABLE) in RAM and the next word is the length of this table. All of these locations are referenced in the CAPS-6 microcode and therefore their definition can not be altered. The INT.TABLE occupies locations 8 to 16. It contains vectors to various interrupt handling processes. Each of these processes has a PSD associated with it. Additionally, there are eight other processes as defined in Figure 3-8. PSD's associated with each of these processes (with the exception of application task PSD's which reside in the system memory) reside in the PROM locations 18 to 9F. The PROM copies of the PSD's are used only to initialize the RAM copies of the PSD's upon system restart or after being reset. The PSD's in RAM are updated when a process terminates or is interrupted. In the PROM version, the PSD pointer and LENV are null, indicating the uninitialized state of the process.

Figure 3-14 shows the Kernel data bases that reside in the first four pages of the cache RAM. The first word of RAM (2000) points to the active PSD (APSD). The next four words are used by the COOPERATOR program to communicate with the CTA (see Section 4.1). The next three words are the alternate program vector (SPCR, TOS, and STKLIM). Six words starting at 2008 are reserved for the Kernel PSD. The next word is the page fault word. It contains the virtual address that caused the last

Address	Content
0-2	Kernel vector (SPCR, TOS, STKLIM)
3	Kernel PSD Pntr
4	PM Table Pntr
5	PM Table Length
8-16	Interrupt Table
	Interrupt PSD's
18-1F	No Interrupt
20-27	Overflow
28-2F	IPC
30-37	Timer
38-3F	Write Protect Violate
40-47	Page Fault
48-4F	Test Adapter
50-57	Illegal Opcode
58-5F	Stack Overflow
60-67	NonLocal Search Fault
68-6F	Privileged Instruction Fault
70-77	Privileged Mode Violate Fault
	Other PSD's
78-7F	Restart
80-87	R4 Dispatcher
88-8F	R3 Dispatcher
90-97	R1 Dispatcher
98-9F	Idle

Figure 3-13. Kernel data base, Cache PROM.

Address	Content
2000	APSD Pntr
2001-2004	Test Adapter - COOP
2005-2007	Alternate Program Vector
2008-200D	Kernel PSD
200E	Page Fault Word
2010-201F	PM Table
	Interrupt PSD's
2020-2027	No Interrupt
2028-202F	Overflow
2030-2037	IPC
2038-203F	Timer
2040-2047	WP Violate
2048-204F	Page Fault
2050-2057	Test Adapter
2058-205F	Illegal Opcode
2060-2067	Stack Overflow
2068-206F	NL Search Fault
2070-2077	PI Fault
2078-207F	PM Fault
	Other PSD's
2080-2087	Restart
2088-208F	R4 Dispatcher
2090-2097	R4 Task
2098-209F	R3 Dispatcher
20A0-20A7	R3 Task
20A8-20AF	R1 Dispatcher
20B0-20B7	R1 Task
20B8-20BF	Idle
	Stacks
20C0-20DF	Kernel
20E0-20EF	Stack Overflow
20F0-20FF	NL Search Fault
2100-2153	Restart
2154-2173	Idle
2174-21FF	R4 Dispatcher
2200-2267	R3 Dispatcher
2268-22EF	R1 Dispatcher
22D0-22DF	Illegal Opcode
22E0-22EF	Stack Overflow
22F0-22FF	WP
2300-2327	IPC
2328-234F	Timer
2350-237F	Page Fault
2380-23B7	No Interrupt, TA
23B8-23C7	PI Fault
23C8-23D7	PM Fault

Figure 3-14. Kernel data base, Cache RAM.

page fault. Sixteen words starting at location 10 are reserved for the PMTABLE. Each entry of this table points to a privileged mode routine that can be accessed only if the process calling the routine is privileged. The microcode for the instruction PMCALL checks for this requirement and creates a PM violate interrupt if the condition is not satisfied. Locations 20 to BF are reserved for PSD's of all the processes including the applications task processes. They are arranged according to their built-in priority which was described earlier in Section 3.1. Initially they do not contain any useful information. The Restart PSD is initialized from the PROM by the Kernel before transferring control to Restart process. The Restart initializes all the remaining PSD's (except applications task PSD). Initialization of applications task PSD's is done by the dispatcher and will be described in Section 3.4. The next three pages of the cache memory (20C0 to 23D8) are allocated as stack space for all processes (except applications tasks). One restriction for allocating this stack space is that stack area for a process should not cross a cache page boundary. This makes sure that a process doing data transfer between cache and the system memory would not inadvertently cross a cache page boundary. Cache or system memory page boundary can not be crossed in a single system bus transaction. To do a bus transaction that involves more than one page, the transaction must be subdivided. This will be further elaborated upon in Section 3.7 which describes the system bus access service routines. For the sake of completeness it might be noted here that one page (256 words) is allocated to each application task as its stack space. Since up to three tasks,

one from each rate group, can be in progress simultaneously, three cache pages are reserved for application task stacks: 2500-25FF for R1, 2600-26FF for R3, and 2700-27FF for R4 rate group tasks (see Section 3.4 for an explanation of these terms).

3.3 System Restart

3.3.1 Functions

The Restart program has two functions: one, to bootstrap the system initially, also called the "cold start," and two, to restart the system after a power interruption, also called the "hot start."

In the hot start, the system memory is already loaded with the required program and data, the system configuration tables are already initialized (with the configuration of the system when the power went down) and the task of the Restart program is to bring the system back up to this configuration, keeping in mind the possibility that everything that was operational before the power interrupt as indicated in the configuration tables may no longer be so, that is, a unit may have failed during the power transient. It may be recalled here, by the way, that the system memory as well as all the system control unit registers within each LRU have battery back-up while the cache memory does not. Therefore they can be considered non-volatile from the power transient viewpoint. The Restart program assumes that the system memory and control register contents are unaltered with the exception of a failed unit. A unit may have failed or an LRU may be uninitialized. This could, for example, happen if the system power was shut off and an LRU was replaced by a new

LRU (on the ground, of course). After a hot start, a number of initialization functions must be performed. For example, since the cache RAM has no back-up,the mapper table in each LRU must be cleared to indicate that none of the pages are loaded in cache. No effort is made to restart in midstream tasks that were in progress when the power went down. Instead all the task queues are reset so that task dispatching may begin afresh. The process PSD's in the cache RAM are also, of course, lost and these are reinitialized from their PROM versions. The new processes are, however, not chained together. Restart must also perform this function.

In the case of the cold start the system memory does not contain any useful information. Additionally, the control registers in the LRU's are all zero as well. The system memory is first bootstrapped from an external device much as a cassette tape on a 1553 remote terminal. Once the system configuration tables in the system memory have been initialized, the rest of the system start functions are the same here as in the hot start.

Whether it is a cold or a hot start, there should be one and only one processor triad that executes this program. Also, this is where members of each processor triad come together and synchronize themselves since the processors would lose synchronization once the power is turned off and the power would not come back to all the LRU's at precisely the same instant. The two basic and important functions of the restart program, then are to synchronize members of each processor triad and to arrange communication between processor triads such that any processor triad can bring the whole multi-processor back to the configuration

indicated by system configuration tables and that triad is not hindered by the remaining processor triads. This is accomplished as described in the following.

As indicated in Section 3.2, upon application of power each processor executes the Kernel program which initializes the Restart PSD and passes control to the Restart process. In this process all interrupts are masked. This process runs in privileged, unmapped mode since the page fault process has not been initialized at this point. Every processor that was active prior to power interruption (its run/reset bit was in run mode) would start up as soon as the power is applied and end up in the Restart process. After processor and cache initialization (details of which will be described shortly in subsection 3.3.3), the processor checks its triad identification contained in CPU Register R0. If it is valid (not zero) the processor tries to synchronize with the other two members of its triad. The first step here is to do a dummy system bus transaction. Since at least two members are required to access the system bus, the first processor to arrive at this instruction is forced to wait (by hardware) until joined by at least one more member of the same triad. Once two out of three members have synchronized thus, they repeat the synch loop by doing dummy bus transactions a predetermined number of times which is long enough to account for variations in power being applied to different LRU's. After each such loop the count is transmitted by the triad member to themselves through the IPC register so that the third member would bring its loop counter in agreement with the other two processors. The system bus is hogged at the end of this loop

so that the first triad to synchronize has complete control over the system. This triad then reads system configuration tables from the system memory and brings the system up to the configuration. (Details of this are describedin Section 3.3.3.)

Having restarted the system, this processor triad (which once again appears as a single processor in a multiprocessor system) pends R4 Dispatcher process under the Restart process. Restart then does a resume which transfers control to the dispatcher. If this triad or any member of it needs to synchronize themselves with a new processor (as would be the case when one of the members is being replaced) they would return back to Restart process. However, they would start this process where they left off, which is just after Resume. Here the new member is reset, issued the new triad ID and synchronized with the other triad members.

One last function performed by the Restart program is recognition of the master LRU mode. When the system is being started up with a master LRU for debugging purposes, the Restart program activates the COOP process thereby passing control to the test adapter. It regains control only when COOP does a resume. In this mode, though, other processors are not started so that the Executive software may be debugged on a single processor.

Section 3.3.2 describes the data bases required by this process. Section 3.3.3 describes the program in detail using N-S flowchart of Restart.

3.3.2 Data Base

The most important data sets used by the Restart process are the system configuration tables. There are five such tables. They reside in the first page (0000 - 0100) of the system memory. Their structure is detailed in the following.

(1) TRIAD ID TABLE. This is a 12 word array. The nth entry in this table describes the status of the processor in the nth LRU, that is, whether it is a member of a working triad, a spare, shadow, or a failed unit. If it is associated with a triad, its triad ID is also contained in this word. The exact structure of each such entry is shown in Figure 3-15.

(2) MEMORY RELOCATION & RTC TABLE. This is a 12 word array. The nth entry in this table describes the status of the system memory in the nth LRU, that is, whether it is a member of an active triad, a spare, shadow, or a failed unit. In any case, the relocation assigned to this memory (Memory Relocation Register) is also contained in this word. One bit in each word shows the status of the real-time clock in that memory unit, that is, whether it is armed or disarmed. Figure 3-16 shows the exact structure of each entry in this table.

(3) P, R & T BUS ENABLES. This is a 12 word array. The nth word shows the P, R and T buses on which the nth LRU is enabled to transmit. Figure 3-17 shows the structure of such a word.

(4) C BUS ENABLES, CLOCK STATUS AND T SELECTS. This is a 12 word array. The nth word shows the status of the clock element in

TRIAD.ID.TABLE: 12 WORD ARRAY

ITH WORD HAS TRIAD ID OF ITH LRU AND STATUS OF THE LRU.
THERE ARE TWO STATUS BITS WITH THE FOLLOWING INTERPRETATION:
0 = SPARE
1 = ACTIVE
2 = FAILED

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	X	X	X	X	X	X	LRU STATUS		X	TRIAD ID OF LRU		

Figure 3-15. Triad ID table.

MRR.TABLE: 12 WORD ARRAY

ITH WORD HAS SYSTEM MEMORY RELOCATION VALUE (BITS 0 TO 4) FOR ITH LRU, ARM/DISARM BIT (BIT 5) FOR THE REAL TIME CLOCK AND STATUS OF MEMORY MODULE (BITS 6 & 7) IN THE ITH LRU. CLOCK IS ARMED IF BIT 5 IS '1'. MEMORY MODULE IS A SPARE IF STATUS BITS ARE '00', ACTIVE IF '01' AND FAILED IF '10'.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	X	X	X	X	MEMORY STATUS		RT CLOCK	RELOCATION CONSTANT				

Figure 3-16. Memory relocation and RTC table.

PRT.TABLE: 12 WORD ARRAY

ITH WORD HAS P, R AND T BUS ENABLE BITS FOR ITH LRU.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	P ENABLES					R ENABLES					T ENABLES				

Figure 3-17. P, R, and T Bus enables.

the nth LRU, that is, whether it is a member of the clock quad, a spare, or a failed unit. The C buses on which that clock element is enabled are also shown. In addition, the T bus select code for BGU's of the nth LRU is also contained in the nth entry of this table. Figure 3-18 details the structure of this table.

(5) P, R, T & C BUS SELECT TABLE. This is a 12 word array. The nth entry contains four 3-out-of-5 bus in-mux codes (bus select codes) for the nth LRU, one each for P, R, T and C buses. Figure 3-19 shows the structure of this table.

The reader should refer to FTMP.ASM(TABLES) for more information about the initial system configuration and the content of these tables.

There are four other words, three in the system memory and one in cache RAM, that are used by the Restart program.

(1) RESTART FLAG. The triad that starts up the system sets this flag in the system memory to indicate the fact that the system was restarted. This flag is used by the task dispatcher to reset the task queues and perform other initializations made necessary by the system being restarted (see Section 3.4).

(2) TRIAD VOLS. This word in system memory indicates which processor triads have volunteered for task assignment. Each triad that successfully goes through the synchronization process sets a bit (determined by its triad id) in this word. This word is used by the dispatcher to assign tasks (see Section 3.4).

TC.TABLE: 12 WORD ARRAY

ITH WORD HAS C BUS ENABLE BITS (BITS 0 TO 4) FOR ITH LRU, STATUS OF OSCILLATOR IN THAT LRU (BITS 6 & 7) AND T SELECT BITS FOR BGU'S OF THAT LRU (BITS 8 TO 11).

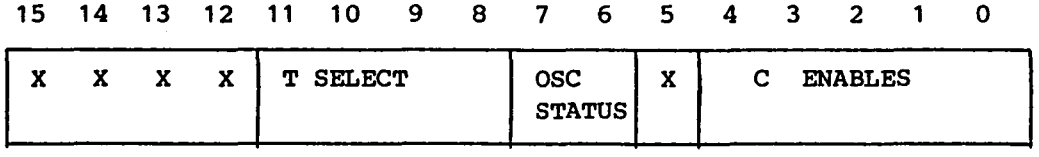


Figure 3-18. T Selects and C enables.

SELECT.TABLE: 12 WORD ARRAY

ITH WORD HAS P, R, T AND C BUS SELECT BITS FOR ITH LRU.

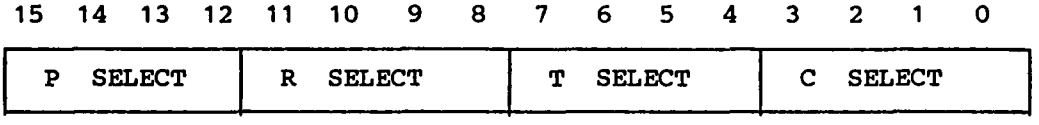


Figure 3-19. P, R, T, and C Select table.

(3) SYNCH COMMAND. When a triad returns to the Restart process for synchronization with a new processor member it checks this word in system memory to obtain the LRU number of the new member. The synch command also contains a bit that is set to 1 if this is the only triad in the system. If so, the triad pends R4 dispatcher and resumes dispatching tasks. If not, it drops down into idle process and waits for an IPC interrupt from other triads to start R4 dispatcher. This command is written by the configuration controller. The triad indicates the completion of the synchronization process by clearing this command in the system memory.

(4) TRIAD ID. This global variable in cache is set by the Restart program for use by other programs. Each processor triad is assigned a 3-bit triad identification. This is stored in CPU control register R0. The triad ID can be any number from 0 to 7. However, the leading bit of the triad IDs should be 1 in order for the bus poll hardware to function properly. This eliminates the use of triad IDs 0 to 3. Therefore triads are assigned numbers 5, 6, or 7. However, since triad ID is used as an index for a number of data sets, the global variable TRIAD ID is set to 1, 2, or 3 in place of real triad IDs 5, 6, and 7, respectively.

3.3.3 Restart N-S Diagram

This AED program resides in the cache PROM of each LRU. The source code for this program is in FTMP.AED (RESTART) and the compiler

output is in FTMP.LIST (RESTART). A flowchart for this program is shown in Figure 3-20. As indicated earlier, Kernel transfers control to this program after a processor reset or a configuration control induced reset. In either case, the first function of this program is to initialize a number of items in the processor and the cache memory. The processor items are initialized by a subroutine called INITIALIZE.PROC. This routine resets system bus access control register, clears the interval timer in the processor, clears all interrupts and initializes the memory mapper. Status of the PROM (32 pages) and lower 5 RAM pages (2000 - 24FF) is set to "Present" and "write protected." The user stack area (2500 - 27FF) is set to present and the remaining virtual address space is set to "not present." The mapper and I/O pages are initialized as present and write protected. The cache initialization is done by a subroutine called RESET.KERNEL. This procedure initializes all the PSD's in the RAM (except Restart and user task PSD's) by copying them from PROM. It also clears the user stack. Since this procedure references a number of data bases defined in Kernel it is assembled with Kernel. Having initialized all the PSD's, it pends idle process to the Restart process.

The next step is to form a triad by synchronizing with the other two processor members of the triad if the processor has a valid triad id. This is done by the synch procedure as explained in Section 3.3.1. Once a triad has successfully gone through this synchronization process it is holding the system bus. Therefore any other triad that is in the middle of the synchronization process would wait until the bus is released. The

RESET SYSTEM MEMORY CONTROLLER.
RESET INTERVAL TIMER.
CLEAR ALL INTERRUPTS.
INITIALIZE MAPPER.
INITIALIZE PSD'S IN CACHE RAM.
PEND IDLE PSD.

READ TRIAD ID FROM CPU REGISTER 1.

IS THE TRIAD ID VALID? (NOT ZERO)

YES	NO	
HOG BUS. I=15	WAIT 5 MSECS. READ TRIAD ID.	
WHILE I > 1 (SYNCHRONIZE WITH 3RD CPU HERE.) SYNCH(). (ALLOW 3RD CPU TO GET IN HERE.) XMIT I TO SELF IPC REG1. READ IPC REG1. SET I = IPC REG1 - 1.	IS TRIAD ID = 0?	
	YES	NO
	WHILE TRUE COLD START. (BOOT FROM TERMINAL.)	FALL THRU & SYNC. WITH A TRIAD.
READ MEMORY RELOCATION (MRR) VALUES FROM SYSTEM MEMORY AND REISSUE MRR'S TO ALL LRU'S. REISSUE ALL P,R,T AND C BUS ENABLES AND T SELECTS TO ALL BGUS. REISSUE SELECT CODES TO ALL LRU'S READ TRIAD IDS OF ALL LRU'S FROM SYSTEM MEMORY. HALT ALL PROCESSORS EXCEPT MEMBERS OF SELF TRIAD. ISSUE '0' TRIAD ID TO ALL PROCS. RESTART ALL PROCS EXCEPT SELF AND BAD PROCS. WAIT UNTIL ALL PROCS ALL PAST THE 'POWER ON RESET' DECISION POINT. REISSUE ALL GOOD PROCS TRIAD IDS. CLEAR REAL TIME CLOCK. SET RESTART FLAG TO TRUE IN SYSTEM MEMORY. REFRESH SYSTEM MEMORY TO BRING ANY NEW LRU BOX UPTO DATE. CLEAR ALL ERROR LATCHES. XMIT 'PEND R4' REQUEST TO SELF. RELEASE BUS.		

Figure 3-20. Restart N-S diagram.

WHILE TRUE

HOG BUS.
DO A DUMMY BUS TRANSACTION.
RELEASE BUS. (SYNCHRONIZE WITH 3RD CPU.)
ENABLE IPC INTERRUPTS. (ALLOW R4 TO BE PENDED HERE.)
RESET CONFIGURATION CONTROL COMMAND TO ZERO.
UPDATE TRIAD STATUS WORD IN SYSTEM MEMORY TO INDICATE TRIAD AVAILABILITY.
IF ONLY TRIAD IN THE SYSTEM THEN PEND R4.
RESUME.

READ CONFIG. CONTROL COMMAND. (RETURN HERE TO SYNC WITH AN LRU.)

IS THIS A 'SYNCHRONIZE' COMMAND?

YES	NO
RESET TARGET LRU. XMIT '0' TRIAD ID AND RUN COMMAND TO THIS LRU. CLEAR UNUSED CACHE TO 0. INITIALIZE MAPPER, TIMER, SBA CNTL ETC. WAIT TO ALLOW TARGET LRU TO COMPLETE ITS INITIALIZATION. XMIT TRIAD ID AND RUN SIGNAL TO TARGET LRU. WAIT TO ALLOW TARGET LRU TO GET TO BUS POLL.	N I L

Figure 3-20. Restart N-S diagram (cont.)

bus, however, is held by the controlling triad unit. It has reset all the processors. This forces the competing triad members to start over at the beginning. In any case, once a triad has been formed, the system configuration tables are read from system memory into cache and the system configuration is brought in conformity with the tables. MRR (memory relocation including RT Clock Arm Command) is issued to all LRU's. Each LRU is enabled on the P, R, T and C bus as indicated in the tables. P, R, T, and C bus in-mux codes are issued to all LRU's and T select codes are issued to all BGU's. All processors, except members of the self triad, are then reset and assigned "0" triad id. Of these processors, those marked good are restarted. Since these processors were assigned an invalid triad identification, this time they follow the right hand path of this program once they start executing the Restart program. This ensures that one and only one triad would restart the system without interruption from other triads. A 200-millisecond wait at this point ensures that restarted processors are past the "power-on reset" decision point. At this point all the good processors are issued their correct triad identifications so that they would fall through the right hand side of the program and into the second synchronization phase and form their respective triads. Actually, they fall through and wait for the bus to become available since the master triad is still hogging the bus.

At this point, the master triad clears the real time clock and all the error latches, and sets the restart flag in the system memory. The system memory is refreshed to bring any LRU box up to date. To refresh the system memory it is read into cache one page at a time and written back.

Finally, the R4 dispatcher process is pended into the PSD chain by transmitting a "Pend R4" request to the IPC Register 2 of the self triad. Writing to IPC 2 creates an IPC interrupt which is masked in the restart process. This interrupt stays in the dormant state until it is unmasked further in the program. At that point the routine responding to the IPC interrupt, viz. the IPC interrupt handler, checks the message in the IPC register and pends R4 process in its appropriate place. The IPC interrupt is the standard way of pending R4 process elsewhere in the Executive. Therefore this same method is used here. Having pended R4 process, the master triad releases the bus and falls through into the second synchronization process which is used by all the other processors that followed the right hand path of the program.

The IPC interrupts are enabled here, thereby allowing R4 process to be pended between the Restart and the idle process as dictated by its built-in priority. The IPC interrupt handler which pends the R4 process will be described later in greater detail. The triad sets appropriate bit in the TRIAD.VOLS in system memory indicating it is available for task assignment and then does Resume (0) to terminate the Restart process and start the process pended underneath it. Two possibilities exist. If this was the master triad it would start R4 dispatcher at this point. All other triads, however, would fall into the idle process since they did not pend R4 process. Thus at the end of the system start one triad would start the R4 dispatcher while the other triads wait in idle loop for further instructions from the master triad. After this point the master triad is no more privileged than the other triads. In fact, even during

restart, any triad that happens to arrive at the power-on reset point before all others assumes the role of the prime mover. The term "master triad" has simply been used to distinguish the two paths of the Restart process.

The rest of the Restart program is used to pick up a new member in a processor triad. When the configuration controller decides that a triad should synchronize with a new processor, it issues a triad synch command and a "Pend Restart" request to the target triad. It may be noted here that once a triad does a Resume from restart, the Restart process is terminated and it is no longer in the PSD chain. However, its PSD is updated at the time it is terminated so that when it is pended in the future it would start executing where it left off, that is, right after the Resume instruction. When the target triad returns here it fetches the LRU ID of the target processor from the synch command, resets this processor, issues a 0 triad ID to it and restarts it. The target triad then reinitializes its own processor and cache using INITIALIZE.PROC and RESET.KERNEL procedures. The target triad and target processor then merge together and synchronize. Following synchronization, synch command is cleared and TRIAD.VOLS is updated to indicate availability of this triad once again. If this is the only triad left in the system, as indicated by a bit in the synch command, it pends R4 dispatcher. Otherwise it falls into the idle process.

The Restart program uses a number of procedures that are also used by the Executive elsewhere. These include the system memory access routines such as READ, WRITE, HOG.BUS, RELEASE.BUS and other more specialized

procedures such as CLR.ALL.ELS which clears all error latches. These programs are collectively known as Executive primitives and will be described in detail in Section 3.7.

3.4 Task Scheduling and Dispatching

The user workload in the FTMP consists of repetitive application tasks such as guidance and navigation, autopilot, etc. The Executive tasks such as fault-detection, isolation, and recovery, as well as system displays, also must be executed periodically at certain iteration rates. The task dispatch algorithm, therefore, has been built around this requirement. Three iteration frequencies are provided. A task may be scheduled to run at any one of these three rates which are called R1, R3 and R4. Their nominal frequency at present is 2.5, 10, and 20 Hertz.

Although task scheduling and dispatching is central to any operating system, a number of other peripheral functions must also be performed. The task schedule must be adjusted to take care of temporary system overload. I/O must be performed for tasks that require it. System bus transactions must also be performed for those tasks that are not privileged to do so. All of these functions are performed by the task dispatcher.

Another important part of the Executive is the Configuration Controller. The input to this program is the content of 48 error latches for 12 LRUs. The output of the Configuration Controller is a set of commands to change the system configuration. The Dispatcher executes these commands. Finally, the dispatcher also winds down gracefully the work

assignment of a triad that has been commanded to retire by the Configuration Controller.

All of these functions taken together form the core of the FTMP Executive. Each of these functions will be dealt with in detail in the remainder of this chapter. Section 3.4.1 describes the task dispatch algorithm and its implementation.

3.4.1 Dispatch Strategy and Implementation

As noted previously, tasks can be dispatched at one of three different rates. There is a separate dispatcher program to handle each rate group tasks. The task queues and other data sets are distinct for each dispatcher. Processors are multiprogrammed so that tasks from all three rate groups can be in progress simultaneously. There is, of course, different priority associated with each rate group so that certain tasks can preempt certain other tasks. Specifically, R4 rate group (highest frequency) has the highest priority while R1 rate group (lowest frequency) has the lowest priority. Tasks within a single rate group are treated as equals. They are arranged in a queue and executed in that order on every iteration. Preconditions or constraints for starting a task are allowed so that a certain task may not begin until certain other tasks have completed during that iteration. The iteration cycles of the three rate groups are synchronized so that the beginning of an R1 cycle or frame coincides with the beginning of an R3 as well as an R4 cycle or frame. Evidently, there are eight R4 frames and four R3 frames for every R1 frame as determined by their respective iteration frequencies (see Figure 3-21).

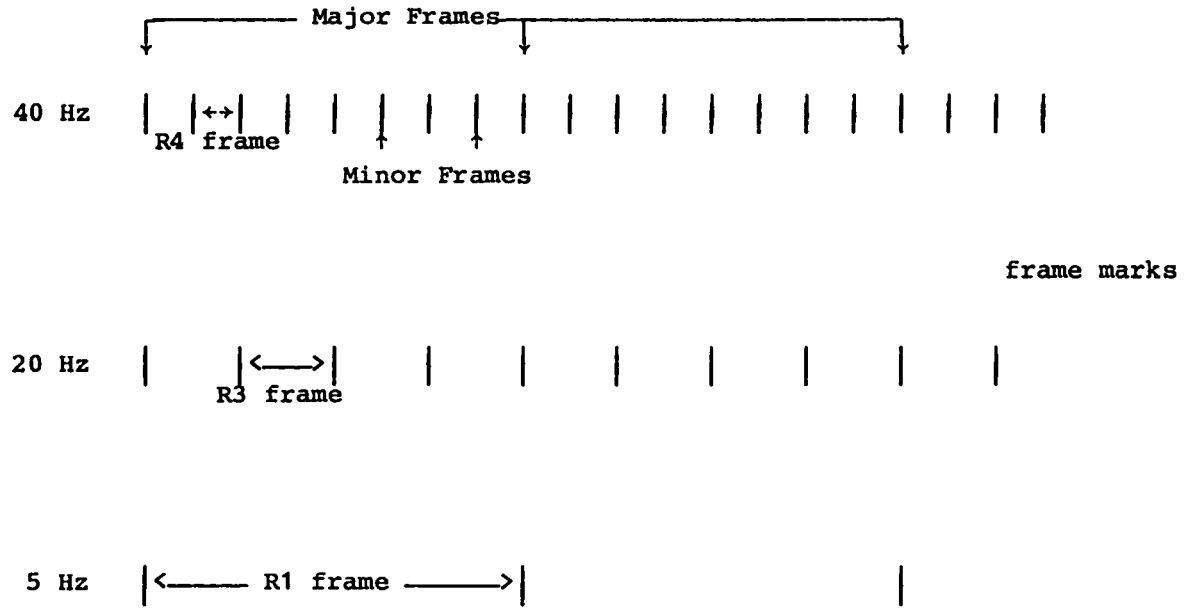


Figure 3-21. FTMP workload characterization.

The start of a task execution cycle in which all rate group tasks begin a new frame is called a major frame. This, of course, happens every eighth R4 frame.

The implementation of the dispatch strategy described is fairly straightforward in a uniprocessor environment. However, in a multiprocessor, a number of additional factors complicate this otherwise simple task management philosophy. For example, factors such as work division amongst processors and resources as well as data sharing must be given consideration. Since these issues only help to cloud the basics of the strategy implementation, it is instructive to first deal with a multiprogrammed uniprocessor environment. The multiprocessing considerations can then be seen more clearly and as a separate issue in their own right. Many multiprogramming details of the uniprocessor implementation, in fact, carry over to the multiprocessor scenario without any change.

With a single processor in the system, the question of workload division does not arise. The sole processor in the system must perform all tasks. Requirement of several frequencies and priorities dictates that the processor be multiprogrammed. A typical implementation would be as follows. There are three dispatchers, one for each rate group. There is a process associated with each dispatcher. The R4 dispatcher process is bootstrapped by the Restart process when the system starts up as described in Section 3.3. This, then, is the start of a major frame. That is, all tasks begin their first cycle at this point in time. Thereafter they repeat as dictated by their respective iteration frequencies. The R4 dispatcher, therefore, pends R3 and R1 dispatcher processes in the

PSD chain. It then performs all the I/O for tasks from all the rate groups and then proceeds with the execution of R4 tasks. R4 task queue is read into cache from system memory and the first task that meets all the required constraints is chosen for dispatch. The pointers in the task queue are rearranged and the task queue updated in the system memory. If the task is not privileged, all the data it needs from system memory is read into its stack area by the dispatcher. The data requirements are contained in a data control block in system memory assigned to such tasks. There is also a PSD block associated with each task that is used to create a process for the task. Having created such an R4 application task process, the R4 dispatcher "activates" this process, thereby passing control to the task. To make sure that the control eventually returns to the dispatcher, a time-out limit, as indicated in the task control block, is placed on each task. The interval timer is loaded with this time so that in case of time-out, a timer interrupt would terminate the task. Upon normal completion of the task, however, it does a Resume to return control to the dispatcher which updates task data blocks in system memory, if required. This process continues until all the R4 rate group tasks have been dispatched. The R4 dispatcher then does a Resume to pass control to the R3 dispatcher. Before doing this, however, it loads the interval timer so that a timer interrupt would occur at the beginning of the next R4 frame. The R3 dispatcher executes all the R3 tasks and passes control to the R1 dispatcher which completes all R1 tasks and does a Resume to fall through into the idle process. Sometime during the R3, R1, or idle process, depending on the work load, it is time to start the next

R4 frame. This is signaled by the timer interrupt. When this happens, control passes to the timer interrupt handler which pends R4 dispatcher in the PSD chain. Whatever process happened to be active at the time, R3 or R1 dispatcher or application task, is suspended while R4 dispatcher starts over. This is a minor R4 frame (see Figure 3-21). Therefore R4 dispatcher does not pend R3 or R1 processes this time. It completes another cycle of R4 tasks and resumes the previously interrupted process after having set the interval timer for the start of the next R4 frame. This R4 frame also coincides with the start of the R3 frame. Therefore, in that cycle, the R4 dispatcher pends R3 process but not R1. Thus the R3 process is pended every other cycle of R4 while the R1 process is pended every eighth R4 frame and this cycle continues to repeat itself.

Two points here need to be elaborated upon. First, the interval timer which is a single hardware device in each processor is to be used for two purposes: one, to place a maximum time limit on each task, and two, to signal the start of each R4 frame. Since the interval timer can only be armed to go off at one point in the future, various timers (up to three of which can be active simultaneously) are tracked in software and the shortest of these is loaded in the hardware timer. The software also keeps track of the identity of the hardware timer so that when it does go off the timer interrupt handler knows which of the four software timers (R4, R3, R1 task or R4 frame) caused the timer interrupt. The second point concerns task scheduling in the face of work overload. It has been tacitly assumed in the previous discussion that the processor is able to keep up with the work load so that when it is time to start, say, an R3

frame, the previous R3 frame is assumed to have been completed. There are, however, situations when for short periods of time the workload exceeds the capacity of the processor. This can happen, for example, during transition from one mission phase to the next, etc. In any case, the dispatch philosophy has been designed to accommodate temporary overloads. Figure 3-22 shows the dispatcher overload performance. The basic strategy here is to not start a new frame until the previous frame is complete. More specifically, if the previous R1 frame is not completed on time, the start of the new R1 frame is delayed by two R4 frames. If the previous R3 frame is not completed on time, the new R3 and R1 frames are delayed by one R4 frame to allow the processor to catch up. Finally, R4 frames are initiated such that at least 40 percent of the processor time is available for R3 and R1 processes. This ensures that an overload in R4 group tasks, which have the highest priority, would not completely preempt the lower priority R3 and R1 group tasks. That is, if in any R4 frame more than 60 percent of the R4 frame time is already consumed by the R4 tasks, then the timer alarm for the next R4 frame is set to go off (0.4 times R4 frame time) in future.

To recapitulate salient points of the multiprogrammed single processor dispatch strategy, each R4 cycle is initiated by timer interrupt. R3 and R1 dispatchers are scheduled by the R4 dispatcher at their appointed times. When all the tasks have been completed, the processor falls into the idle process to be interrupted at the start of the next R4 frame. If tasks are not completed on time, start of future frames is delayed to allow processor to catch up. I/O for all tasks is performed by the R4 dispatcher.

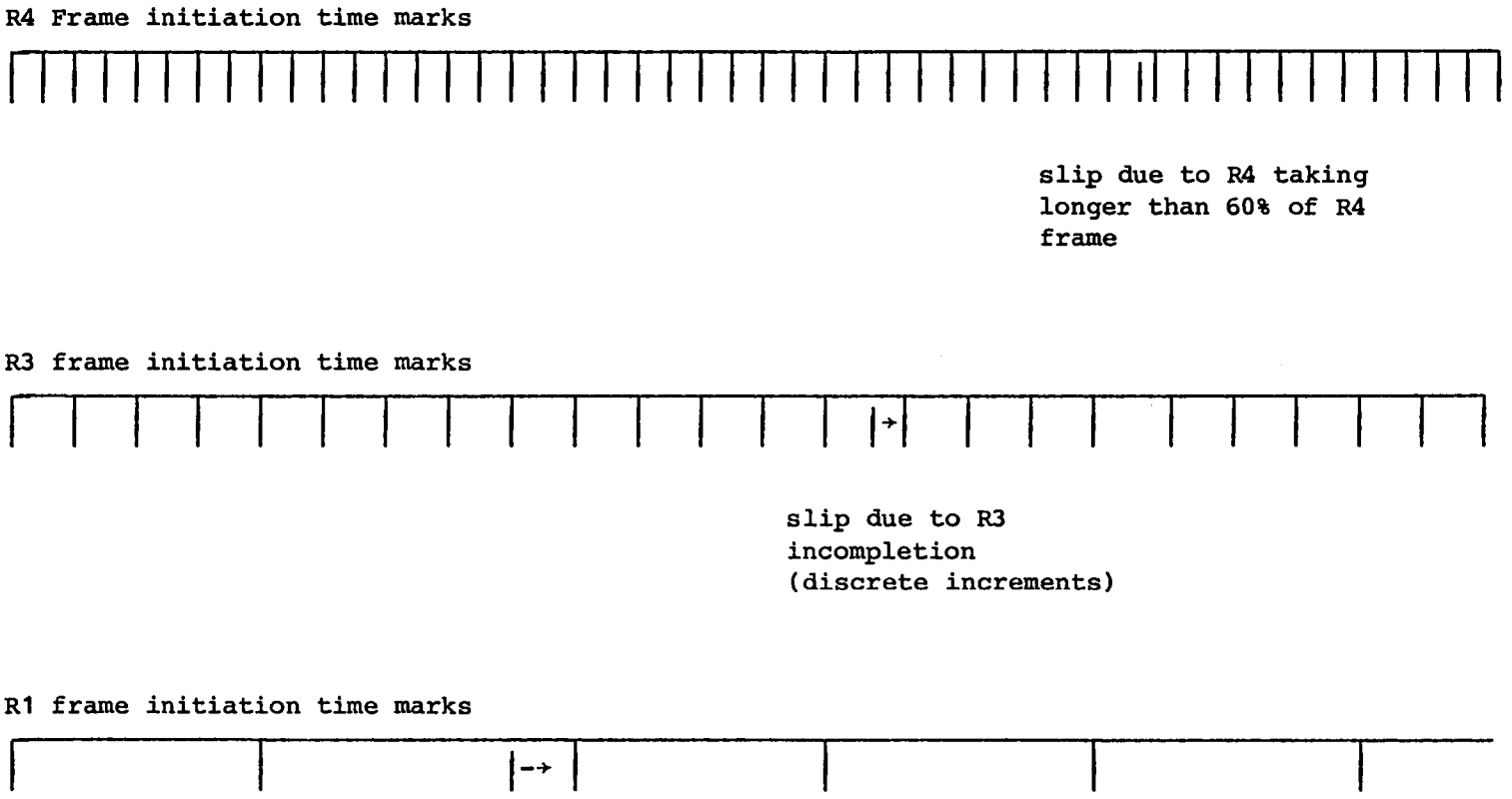


Figure 3-22. Dispatcher overload performance.

This same strategy, with two modifications, can be extended to a multiprocessor environment. First, a timer interrupt is used to initiate an R4 frame in only one processor. This processor then sends a message to other processors through their IPC registers to begin work on R4 frame. Second, since more than one processor can be dispatching tasks from the same queue, it is necessary to provide semaphores to limit access to shared resources to only one processor at any given time. This is accomplished by "lock" and "unlock" functions which control the semaphores. Before a processor can alter a data base it must "lock" it, and after releasing the data base, it must "unlock" it.

With these changes in mind we can follow through a typical task dispatch cycle for the FTMP as illustrated in Figures 3-23 to 3-25. Figure 3-23 shows the initiation of a major frame. Three processors or triads are in the idle state. One processor, A, has its interval timer armed to go off at the start of the next R4 frame. When this happens, the timer interrupt handler pends R4 dispatcher and resumes it. The R4 dispatcher does I/O for R4 tasks. At this point R4 tasks are ready to be dispatched. However, since this is the start of a major frame the R4 dispatcher must also perform I/O for the R3 and R1 tasks. Therefore, processor A "kicks" processor B (sends it a pend R4 request through IPC interrupt). The IPC interrupt handler in processor B pends R4 dispatcher and resumes it. The R4 dispatcher pends R3 and R1, this being the start of a major frame. Meanwhile, processor C continues to be in idle state. Since only one processor can access the dispatcher data bases at any moment only one processor is kicked to start R4. Processor A locks R3

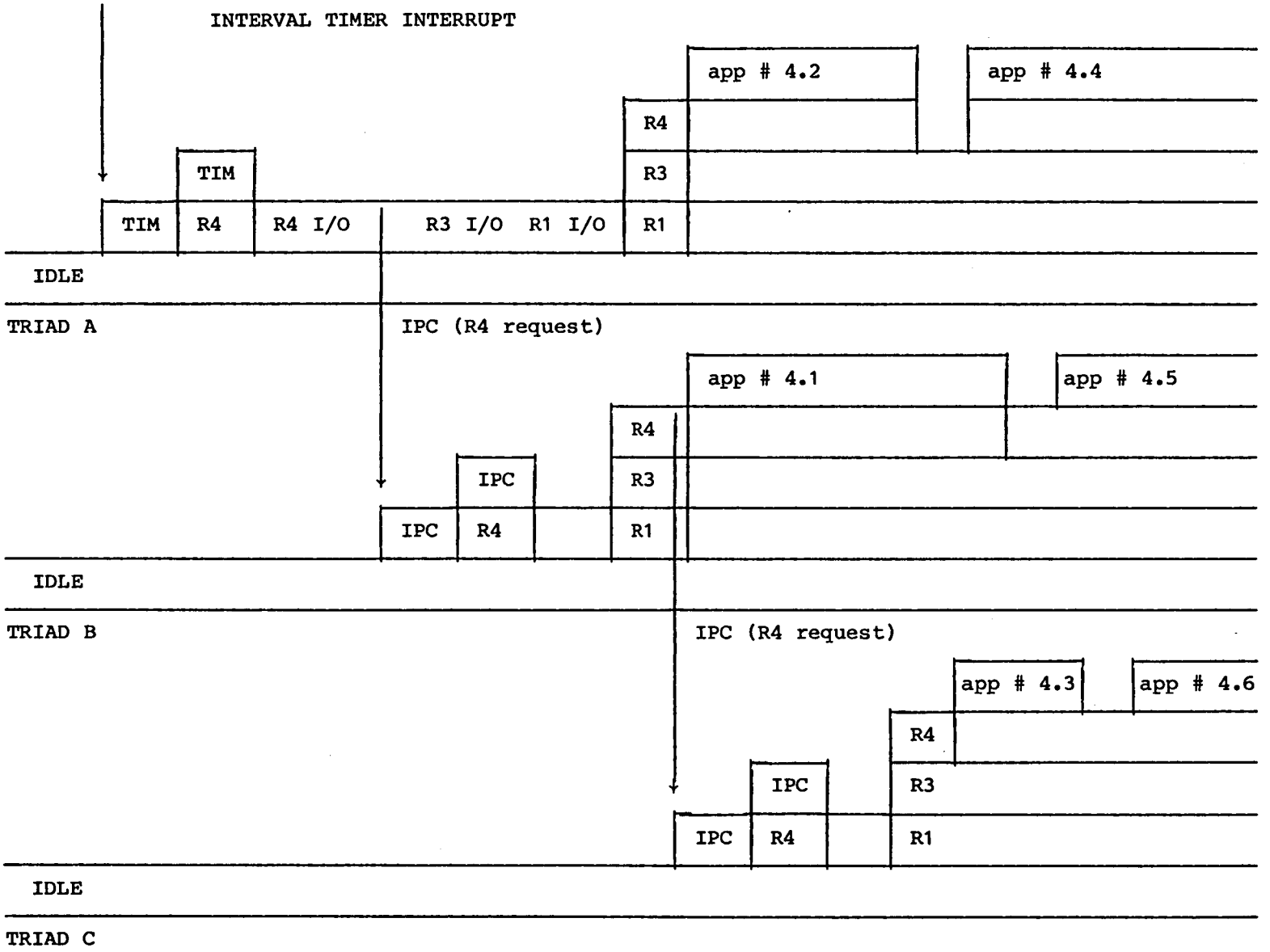


Figure 3-23. Major frame initiation.

and R1 data bases and performs I/O for these tasks. At the same time processor B locks the R4 data bases and searches the R4 task queue and picks up a task to execute. Before starting the task, however, it unlocks R4 and kicks processor C. Eventually all processors start to pick up R4 tasks and execute them.

Figure 3-24 shows how an R4 frame is terminated and R3 resumed. It isseen that processor B is the last to exit from the R4 frame. Processor B, therefore, sets up its interval timer for the start of the next R4 frame and then it resumes the R3 process. Meanwhile processors A and C have already resumed R3 having completed all tasks in R4 frame. Figure 3-25 shows the sequence of events when the timer interrupt occurs in processor B. The situation is now slightly different from that of Figure 3-23. There, (Figure 3-23) all processors were idle. Here,(Figure 3-25) A and B are working on R3 applications tasks with the R1 process pended but yet to start while C is working on R1, all tasks in R3 having already been dispatched. The actions taken here, however, are not much different from the previous case. First of all, B suspends the R3 task and starts the R4 dispatcher. This being a minor frame only R4 I/O needs to be done, at the end of which it picks up a task from the R4 queue and kicks processor A to start R4 work. A picks up the next R4 task and kicks C. The interrupted R3 tasks are resumed when all R4 tasks are completed and the last processor to exit the R4 frame sets itself "R4 responsible" by arming its interval timer.

To summarize the multiprocessor dispatch strategy, an R4 frame is initiated in one processor (R4 responsible processor) by timer interrupt.

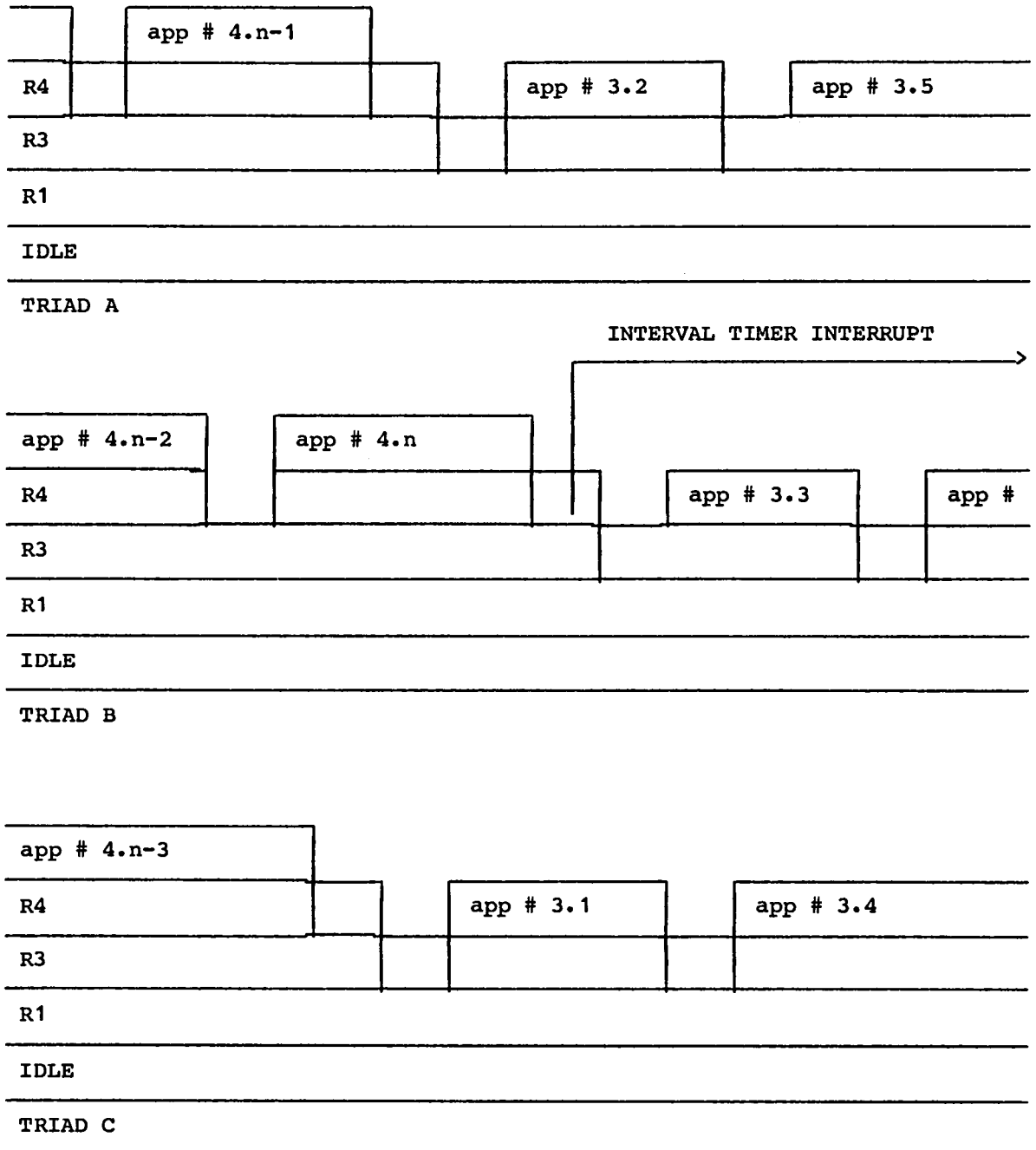


Figure 3-24. R4 Frame termination/Resume R3.

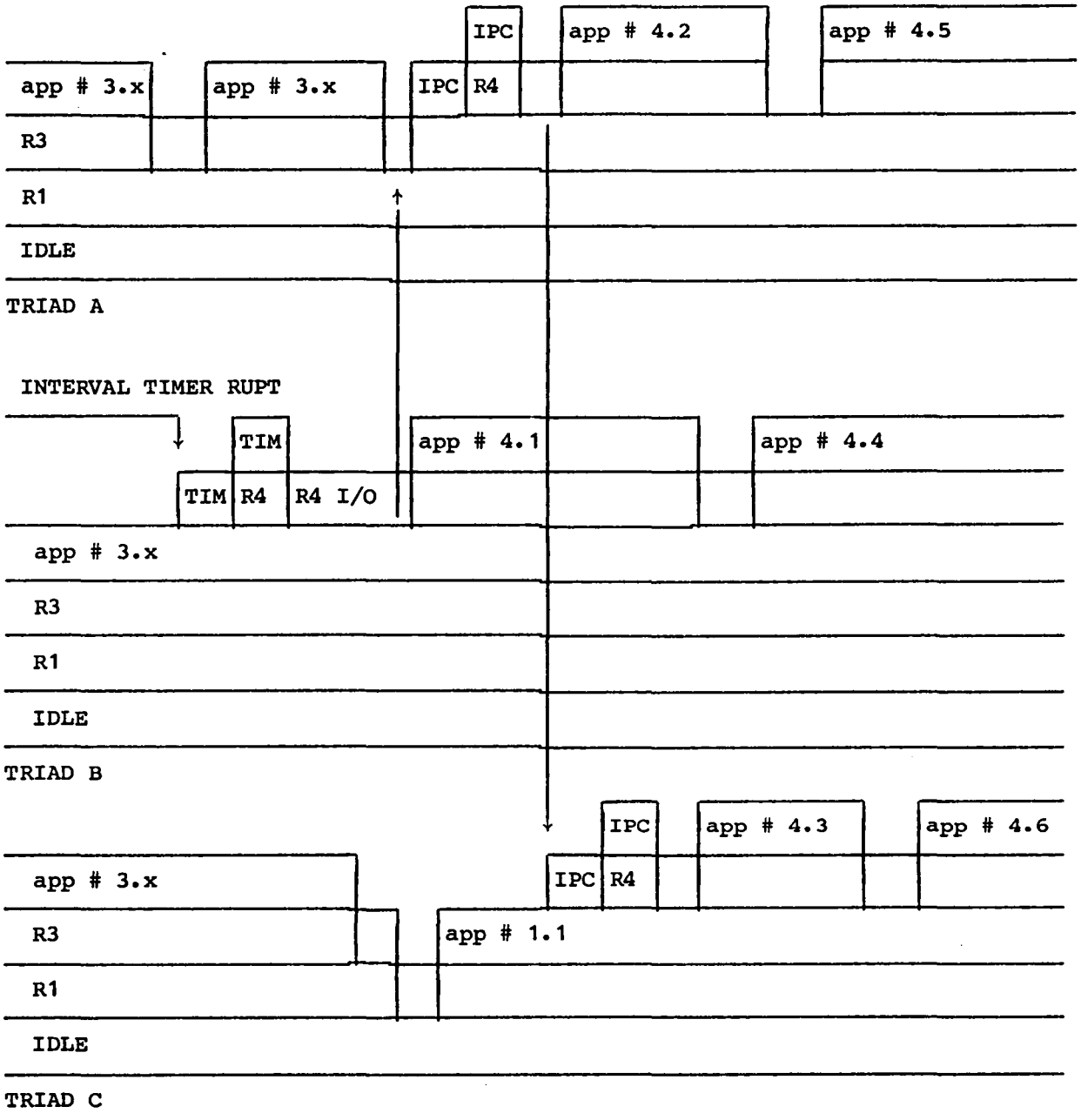


Figure 3-25. R4 Frame initiation, interrupting R3 and R1.

R4 processes in other processors are pended when they receive IPC interrupts from the R4 responsible processor. R3 and R1 processes are scheduled within each processor by the R4 dispatcher. All I/O is done by the R4 dispatcher in R4 responsible processor.

One last complication in the multiprocessor case is the graceful retirement of a processor or triad, as well as the smooth entry of a new triad. This is accomplished with the help of two words in system memory as follows. First, each triad that successfully starts up sets a bit corresponding to its ID in a word called TRIAD.VOLS (triad volunteers) indicating that it is ready for work assignment. When a triad is commanded to retire it resets that same bit after winding down all the tasks it had in progress. The retire command is checked after each task is completed. No new tasks are dispatched by the triad once it sees the retire command. Second, at the beginning of each R4 frame, the R4 responsible triad determines which triads should be assigned work for that frame. It does this by creating TRIADS.AVLBL (triads available) word which is just what TRIAD.VOLS happens to be at the time. This word then is used by the R4 responsible triad to make sure that each triad indicated as being available is kicked at least once and only once to start up the R4 process.

This way a retiring triad gets out as soon as it sees the retire command. On the next frame it is omitted from the work assignment. Similarly a new triad waits in the idle process until the beginning of a new frame. Since the number of triads that started a frame is known clearly, it makes it easy to determine who is the last triad to exit an

R4 frame. This is important since the last triad out is also responsible for initiating the next R4 frame. The system would halt if the last triad out did not set itself R4 responsible. Finally, this also facilitates the determination of the end of an R3 or R1 frame. That is, if all the triads that started the R1 or R3 frame also exited from it then that frame must have terminated. This knowledge of whether the previous frame has ended is central to the decision of starting or delaying the next R3 or R1 frame.

The next two subsections describe the data bases and N-S diagrams of the dispatcher, respectively.

3.4.2 Dispatcher Data Base

This subsection details the data base utilized by the R1, R3, and R4 dispatchers. Although the data bases are unique to each dispatcher, their structure, with some minor exceptions, is identical. Data structures will be described assuming they apply to all rate groups and the differences where applicable will be pointed out.

A majority of the data required by the dispatcher is naturally related to the tasks.

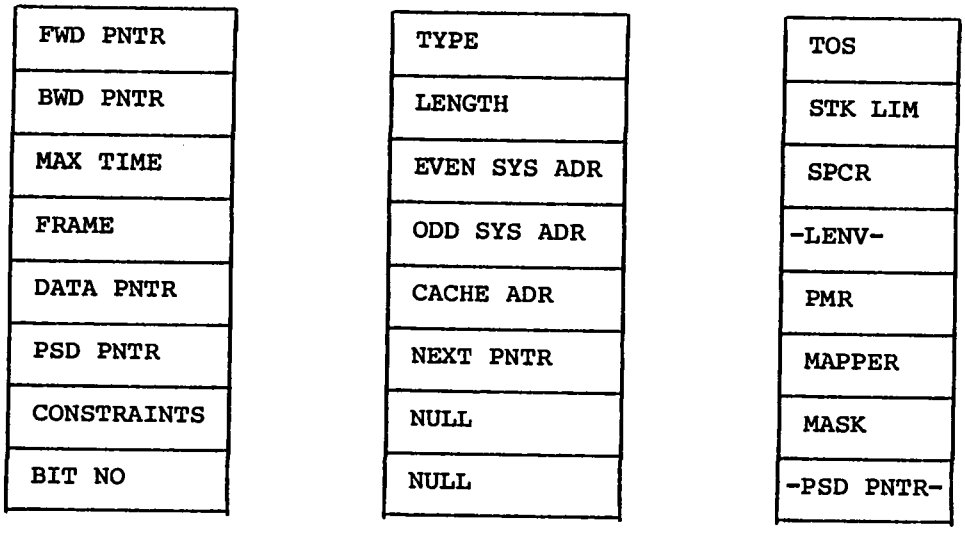
First of all, there is a task queue for each rate group. Associated with each task queue is a control block. Figure 3-26 shows the structure of the R4 control block. There are ten entries in the control block. The first entry points to the beginning of the R4 task queue. The second entry points to the task in this queue that is scheduled to be dispatched next. The third entry is the triad tracker. This word has

TOP PNTR
NEXT PNTR
TRIAD TRACKER
FRAME COUNT
SLIP
TASKS DONE
START R3
START R1
TIME LO
TIME HI

Figure 3-26. R4 control block.

F E D C	B A 9 8	7 6 5 4	3 2 1 0
R4 RESPONSIBLE TRIAD	TRIADS KICKED	TRIADS RUNNING R4	TRIADS TO RUN R4

Figure 3-27. Triad tracker.



(a) Task control block (b) Data control block (c) PSD

Figure 3-28. Task data structures.

four 4-bit fields as shown in Figure 3-27. The four fields indicate i) triads scheduled to run R4, ii) triads currently running R4, iii) triads that have already been kicked to run R4 and iv) the triad responsible for starting the present R4 frame. Of course, the last two fields are not applicable to R3 and R1 control blocks. The next word is the frame number. It ranges from 0 to 15 for R4, 0 to 7 for R3 and 0 to 1 for R1 control block. The next word is "slip" which ranges from 0 to -15. It is used to delay (slip) R3 and R1 frames when they are behind schedule. The next word is called TASKS.DONE. Each task is assigned a bit in this word. When a task completes normally, its associated bit in the word is set. This word is used to determine if constraints, if any, for a task have been met. The next four words exist only in the R4 control block. The first two of these, START.R3 and START.R1, tell the R4 dispatcher whether R3 and R1 are to be pended in this cycle. The next two words constitute double precision absolute start time of the next R4 frame.

Figure 3-28 shows the remaining data structures associated with tasks. The task queue is really a list of task control blocks which are chained together in the forward and backward direction. Figure 3-28(a) shows the structure of one such control block. The top two words are the forward and backward pointers, respectively. The next word is the maximum time allotted to this task. The next word is frame count indicating the last frame number in which this task completed successfully. The next two words point to the other two data sets associated with task, viz., the data control block and the task PSD. The next word, "Constraints," indicates which tasks must be completed before this task is

dispatched. The last word, task done bit number, is the bit number in the TASKS.DONE control word corresponding to this task.

The next data set associated with a task is the data control block shown in Figure 3-28(b). This block or a number of these blocks chained together define data requirements for nonprivileged tasks. It may be recalled here that the nonprivileged tasks are not allowed to access system memory. The dispatcher reads in data before starting the task as defined by these data control blocks. At the completion of the task, results are written back to system memory in accordance with the control blocks. Each control block has a cache address, two system memory addresses and the type and bytes of data and a pointer to the next data block, if any. The two memory addresses are labeled odd and even. These are used on odd and even frame iterations of the task, respectively. If the two addresses are not the same, then that assures the integrity of data and a task can be rolled back and retried without already having destroyed the data. The type of data indicates the direction of transfer, that is, from system memory to cache or vice versa or both.

The last structure associated with each task is a dummy PSD shown in Figure 3-28(c). The structure of the PSD is the same as described earlier in Section 3.1. However, two of its elements, LENV and PSD.PNTR, are null. These are initialized by the dispatcher just before passing control to the task process. Briefly, the remaining elements of the PSD define stack location for the task process, its starting address, privileged/nonprivilege status, whether it was mapped or unmapped and the interrupts that are allowed during execution of this task.

Associated with each task queue is a semaphore or a lock word. This word must be locked before a dispatcher can read in any control bloc from system memory.

There are a certain number of words that are more directly associated with each processor triad rather than each dispatcher. Two of these, TRIADS.VOLS and TRIADS.AVLBL have already been described in the preceding section. TRIADS.AVLBL is used to set various fields of TRIAD.TRACKER in R4, R3, and R1 control blocks. There is also a triad command array, one word for each triad, that is used by the configuration controller to command a triad to retire. Finally, the Restart flag, also described in an earlier section, tells the R4 dispatcher to initialize all the task queues, etc., in the system memory due to a system restart. For ease of experimentation in the FTMP development stages, two other words are provided. One is the R4 Period which is presently set at 50 milliseconds but can be varied. The other is Slop which is set to 40 percent. Slop is the minimum R4 frame time (in percent) allocated for R3 and R1 processes.

All of the variables described reside in the system memory. They are defined in the CAPS assembly language source program FTMP.ASM(TABLES). The assembler output appears in FTMP.LIST(TABLES).

There are also a few cache variables used by the dispatchers. One is EXEC.LEVEL. This is primarily used by the LOCK/UNLOCK routines to avoid deadly embrace. This will be further explained in Section 3.7. EXEC.LEVEL is set to 0, 1 or 2 depending on whether R1, R3 or R4 dispatcher is being executed. The second cache global is R4.RESPONSIBLE which is set to TRUE when a triad arms its timer for the next R4 frame.

Section 3.4.3 describes the dispatcher flow charts.

3.4.3 Dispatcher N-S Diagrams

The three dispatchers, R1, R3 and R4, perform a number of functions which are common to them. The R1 and R3 dispatchers, in fact, are almost identical. They, of course, work on distinct data sets. The R1 and R3 dispatchers may be considered a subset of the R4 dispatcher, from a functional viewpoint. The R4 dispatcher, in addition to doing everything the other two dispatchers do, also initiates new frames, adjusts schedule for overload, performs task I/O, and communicates with the configuration controller. The R4 dispatcher flowchart will, therefore, be described here in detail and only those areas of the R1 and R3 dispatchers that differ in substance from R4 will be highlighted. Figures 3-29 and 3-30 depict the N-S level flowcharts of the R4 and R3 (same as R1) dispatchers respectively. The source programs for the dispatchers may be found in FTMP.AED (DISPR4) and FTMP.AED (DISPR3R1). The compiled listings may be found in FTMP.LIST (DISPR4) and FTMP.LIST (DISPR3R1).

The R4 dispatcher, it will be recalled here, is bootstrapped by the Restart program. On the first pass the dispatcher begins by initializing global variables in cache. R4.RESPONSIBLE is set to false, EXEC.LEVEL is set to 3, R4 period and Slop are initialized by reading these values from system memory and a procedure TIMER.INIT is called that initializes all interval timer related variables. Programs that keep track of various timers in software have been grouped together in one library FTMP.AED(TIMERS). These are explained in detail in Section 3.7.3. In the present discussion only the main function of each timer procedure will be described. Next, the system restart flag in the system

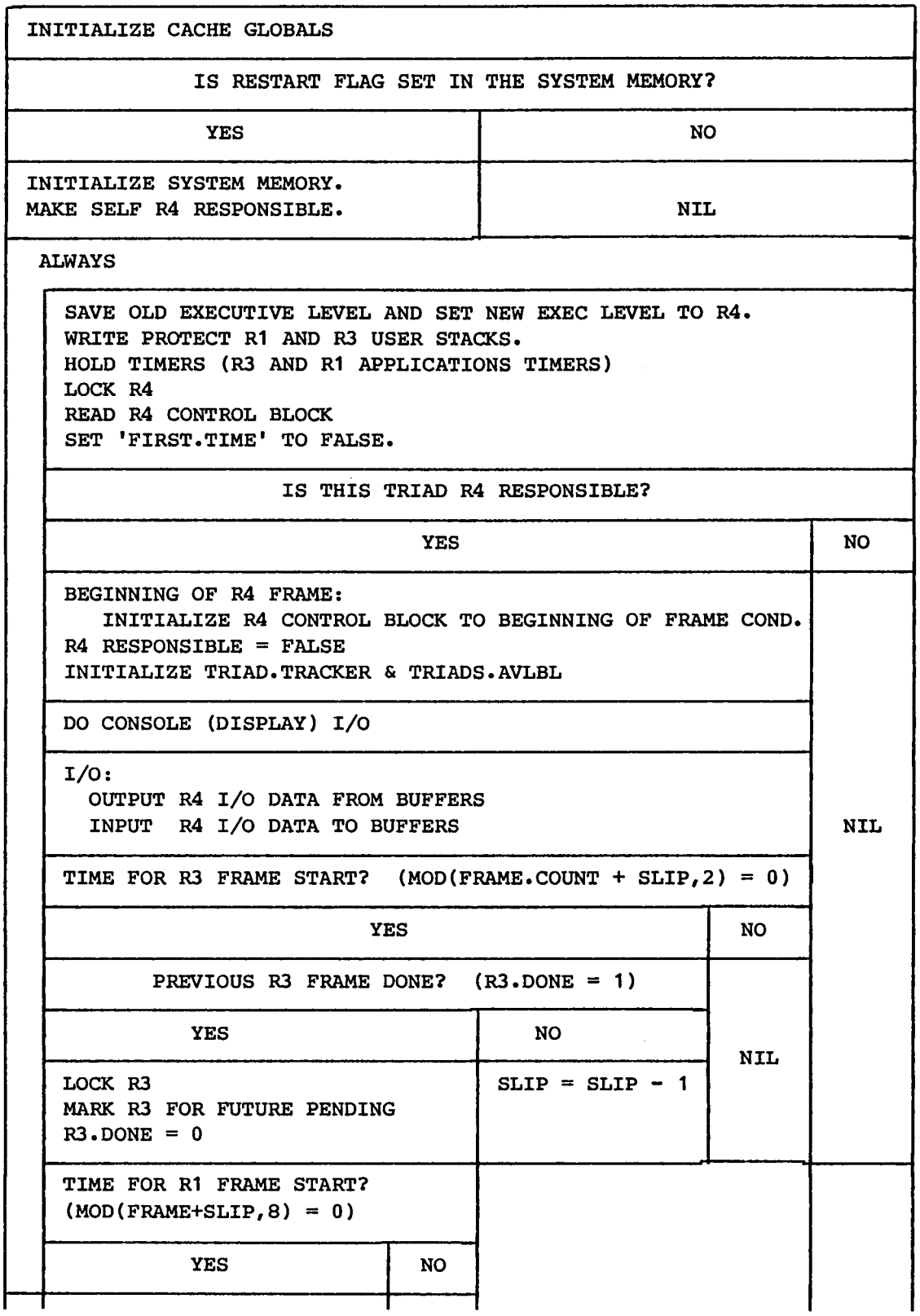


Figure 3-29. N-S diagram of R4 dispatcher.

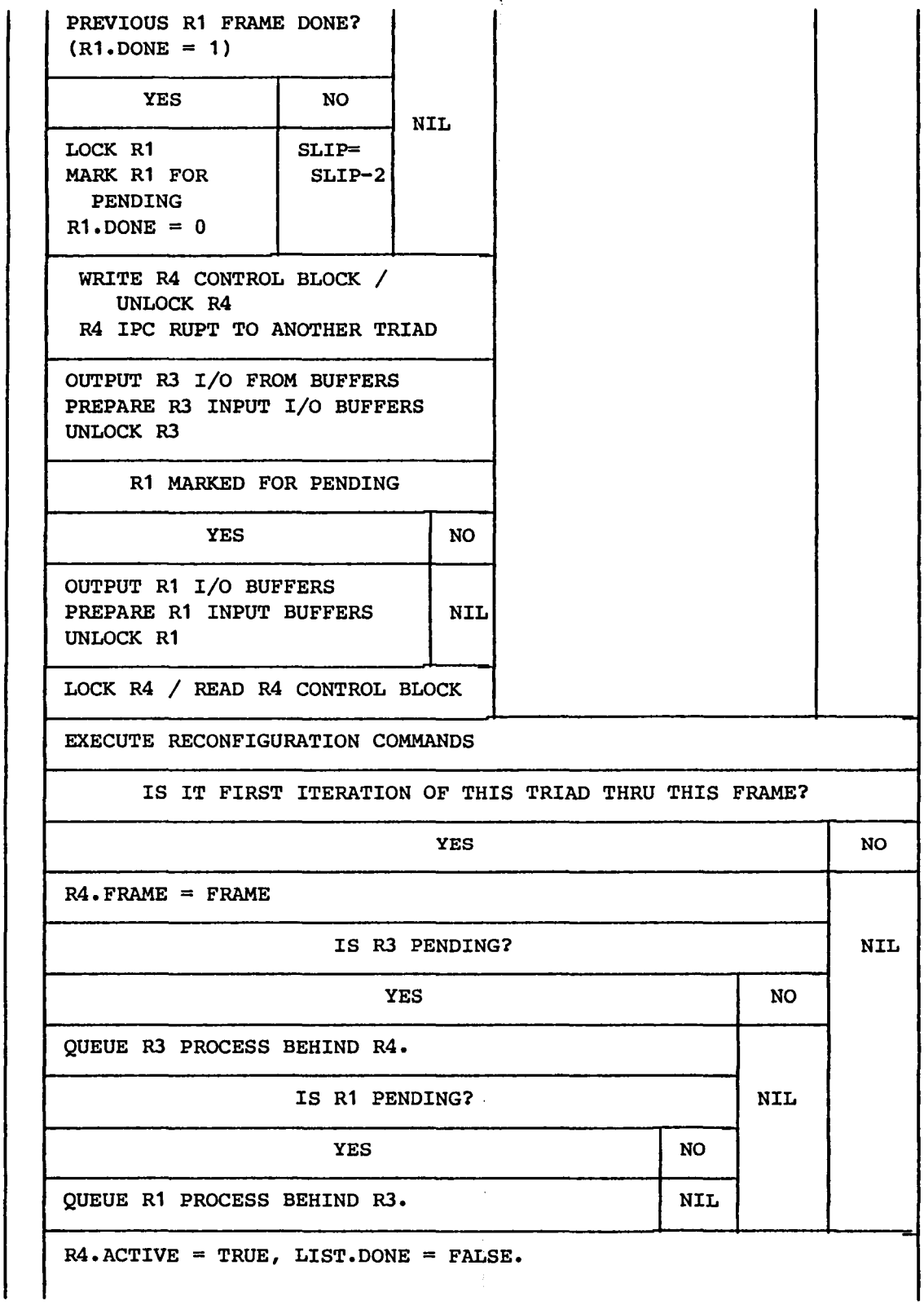


Figure 3-29. N-S diagram of R4 dispatcher (cont.).

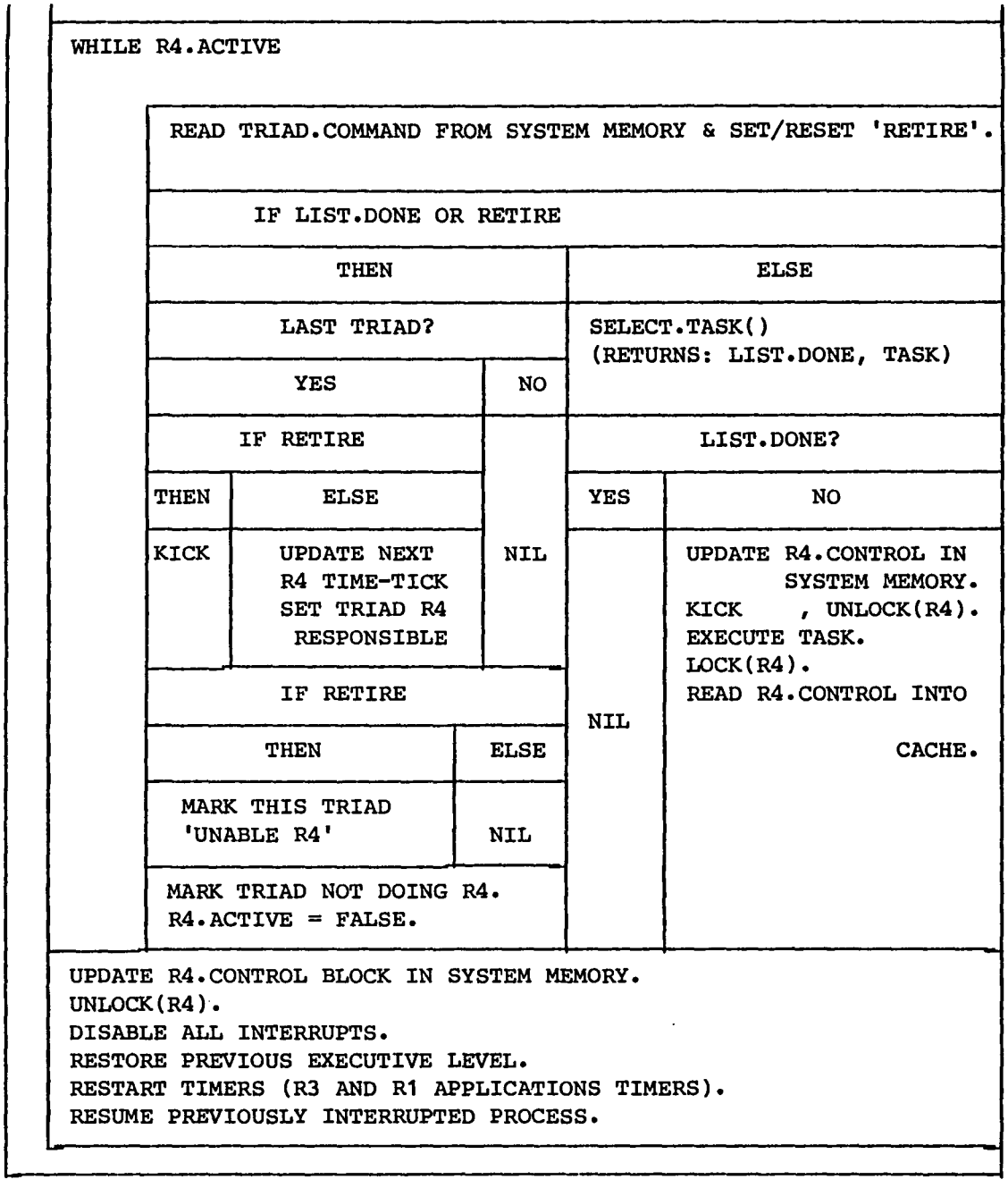


Figure 3-29. N-S diagram of R4 dispatcher (cont.).

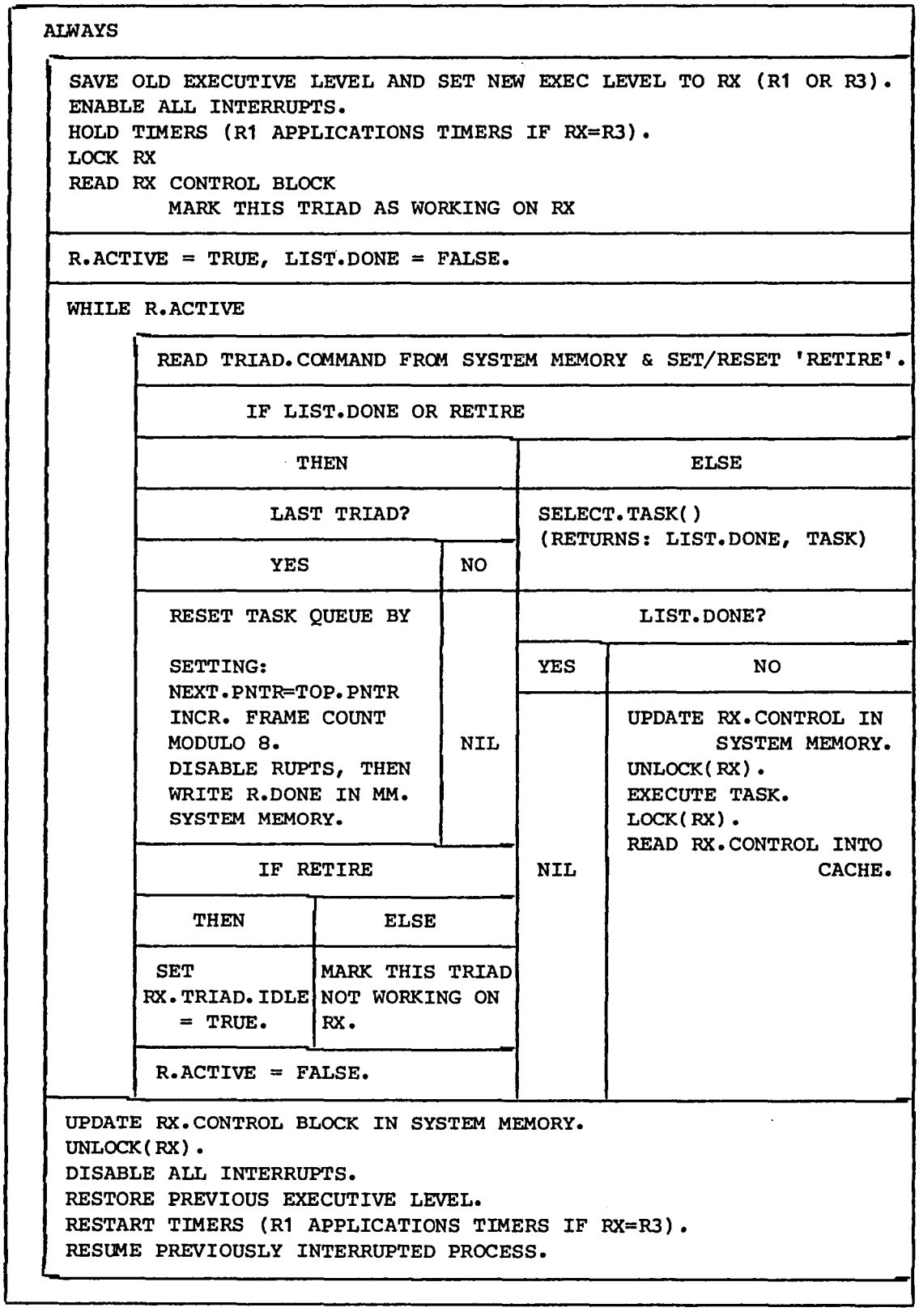


Figure 3-30. N-S diagram of R3 and R1 dispatchers.

memory is checked. If the flag is set, this is the first triad to start up after the system restart. It must, therefore, initialize a number of data structures in the system memory. The restart flag is reset, configuration commands cleared, all resources unlocked, task queue control blocks for all the rate groups reset, configuration controller and 1553 I/O routines initialized. Finally, the triad marks itself R4 responsible. The succeeding triads that start up would skip system memory initialization and also would not be R4 responsible.

The initialization phase is executed only once. The program then falls into an infinite loop from which it can only exit by resuming another process or being interrupted. First of all, a procedure from timers, HOLD.R3.R1.TIMERS, is called. It stops any R1 or R3 task timers which may be running, since a lower priority task may have been interrupted to begin the R4 dispatcher. Next, the executive level (EXEC.LEVEL) is initialized to R4 after saving the previously interrupted execution level, if any. The R1 and R3 user stacks are write protected from the R4 user tasks by marking their status as WP (write protect) in the mapper. The R4 data bases are locked by calling the lock procedure with the R4 semaphore as the argument. The lock procedure would return only when it has locked the semaphore. If the resource is not accessible, that is, it is already locked by another triad, then this triad must wait until it is unlocked. The details of lock, unlock and other general purpose procedures used by dispatcher are described in Section 3.7. Having locked the R4 data base, the R4 control block is read into cache. The FIRST.TIME flag, indicating whether this is the first iteration of this triad though the present R4 frame, is set to false.

Next, if this triad initiated the present R4 frame, that is, if it was the R4 responsible triad, then it must perform a number of frame initialization tasks. The R4 responsible triad sets R4 responsible to false, FIRST.TIME to true. It resets the R4 control block by setting the next task pointer equal to the top pointer, incrementing FRAME.COUNT, setting START.R3 and START.R1 to false, and TASKS.DONE to zero. RECONF.DONE flag, in the system memory, is reset indicating commands for the configuration controller, if any, have not been done in this frame. Next TRIAD.VOLS is copied into TRIADS.AVLBL and various fields of TRIAD.TRACKER initialized from TRIADS.AVLBL. The bit corresponding to this triad in the kick field of TRIAD.TRACKER is reset, indicating this triad should not be kicked to start R4 again. Next, the busy bit in TRIAD.TRACKER is set to indicate this triad has already begun the R4 dispatcher.

The sum of FRAME.COUNT and SLIP is checked to see if an R3 or R1 cycle is also scheduled to start now. If either frame is to be started up, then a check is made to determine if the previous frame for that rate group has finished yet. This is done by checking the first field of TRIAD.TRACKER in the control block of that rate group. It will be recalled here that this field is set at the start of a new frame to indicate all the triads that are assigned work for this frame. As each triad exits the frame it resets its bit in this field. The previous frame has therefore finished if this TRIAD.TRACKER field is zero. If the previous frame is done, then R1.START, R3.START, or both, as the case may be, are set to true. The triad tracker in the R1 and R3 control blocks is also

reinitialized from TRIADS.AVLBL after locking R3 or R1. As other triads, which are not R4 responsible, start the up R4 dispatcher, they check these two variables before pending the R3 and R1 processes. If the previous R3 cycle has not finished the start of the R3 frame is postponed until the next R4 frame. If the previous R1 cycle has not finished, the start of the new R1 frame is postponed by two R4 frames. This is done by decrementing SLIP by 1 or 2 (modulo - 16).

Next, R4 task I/O and Console (FTMP Display and Monitor) I/O is performed by calling appropriate procedures using the DOIT facility of AED. The dispatcher programs reside in the PROM while the I/O programs reside in the system memory and their addresses are subject to change during development stages. Their starting addresses are therefore stored in system memory. The dispatcher reads the starting addresses into its stack and the DOIT procedure calls the program pointed to by TOS. (Normally, the procedure address must be in the instruction stream, that is, the PROM). In any case, if the R3 frame is also to be started then the R4 responsible triad kicks another triad to start up the R4 dispatcher, updates the R4 control block in system memory, and releases the R4 data base by unlocking it. The kick procedure is passed TRIAD.TRACKER as the argument. The kick routine picks a triad that has not been kicked yet and resets that bit in the TRIAD.TRACKER kick field. Having kicked another triad, the R4 responsible triad then performs R3 I/O, unlocks the R3 data base, and performs R1 I/O as well, if this is the start of a major frame.

This completes all the R4 responsible functions. The rest of the R4 dispatcher is executed by all the triads, whether R4 responsible or not.

FIRST.TIME is set true if this is the first iteration of the triad through an R4 frame. The "busy bit" in triad tracker is set to indicate thistriad has entered the R4 dispatcher. RECONF.DONE flag in system memory is checked to see if configuration control commands have been executed. The commands issued by the configuration controller must be executed at least once and only once. Some triads are prohibited from executing some of these commands. For example, a processor triad can not dismantle or reconfigure itself. Any commands that this triad is able to execute, it does by calling a procedure RECONFIGURE. This procedure is logically a part of the configuration controller and will be described along with the controller in the next section.

Next, the R3 and R1 processes are pended in the PSD chain, if appropriate flags are set. Having done this one-time frame initialization, the triad falls into the main loop of the dispatcher to dispatch tasks. It stays in this loop until all the R4 tasks have been dispatched. If LIST.DONE is false, that is, all tasks are not done yet, a procedure called SELECT.TASK is called to pick up the next task from the R4 task queue. An N-S diagram of SELECT.TASK is shown in Figure 3-31. This procedure reads into cache the task control block pointed to by NEXT.PNTR. It checks to see if the constraints for this task, if any, are met. If not, then this task is bypassed. Next, the task control block, pointed to by the forward pointer of this task, is read into

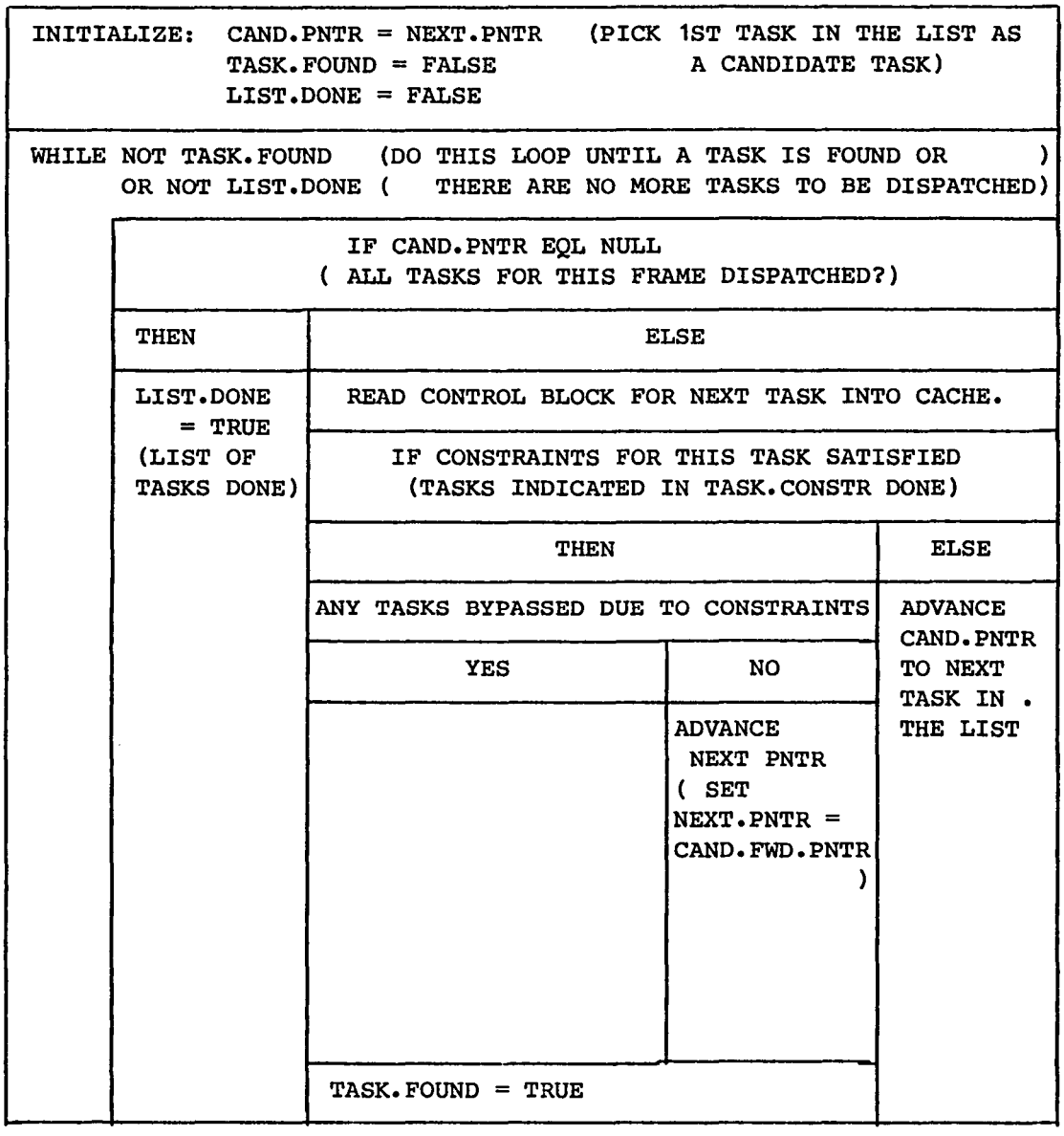


Figure 3-31 Select task N-S diagram

cache. This process continues until a task that meets all the constraints is found. The forward and backward pointers in the task queue are adjusted to reflect the new status, NEXT.PNTR in the control block is updated and a pointer to the control block of the selected task is returned to the dispatcher. If no task is found, LIST.DONE is set to true. Assuming a task is found, the triad kicks another triad to pick up the R4 dispatcher, updates the R4 control block in system memory, and unlocks R4 data base. It then calls a procedure, EXECUTE, to execute the task.

An N-S diagram of EXECUTE is shown in Figure 3-32. This procedure reads in the data control block, if any, of the selected task from system memory. If there is such a block, the required data is read into the indicated cache address from the even or odd memory address, depending on whether this is an even or an odd frame. The data control blocks are followed through their chained pointers until all the required data for this task has been assembled in cache. Next, the task PSD is copied from the system memory into the R4 application task PSD slot. Then a procedure called START.R4.TIMER is called with the maximum time limit for this task as the argument. This procedure loads the interval timer with this time limit. The control is then transferred to the task by activating the R4 application task PSD. If the task terminates normally, control returns to the R4 dispatcher. A routine called STOP.R4.TIMER is called to clear the interval timer. On the other hand, if the task times out, that is, exceeds its time limit, a timer interrupt causes the task process to be suspended and control is transferred to the timer interrupt

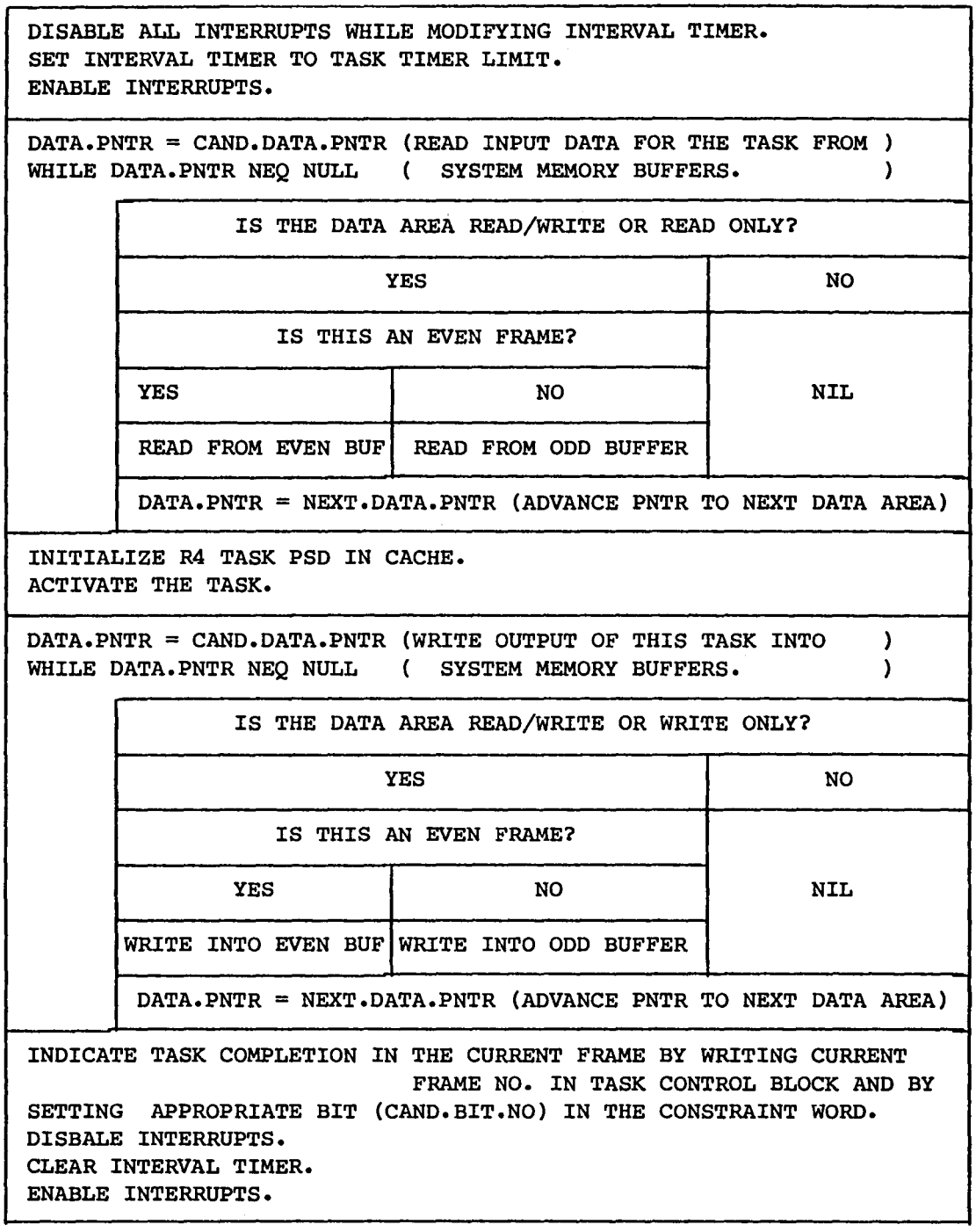


Figure 3-32. Execute N-S diagram

handler. At this point, however, the user task process is still in the PSD chain. The timer interrupt handler, which is also part of TIMERS, having determined the cause of the interrupt purges the user task PSD from the PSD chain, sets a return code in the user PSD and resumes the R4 dispatcher process. The R4 dispatcher checks the task return code and displays a message on the FTMP monitor if the task ended abnormally. Finally, the EXECUTE procedure writes out the task results into the system memory as dictated by the task data blocks. It also updates frame count in task control block to indicate the completion of the task during this R4 frame.

Having executed the selected task, the R4 dispatcher goes back to get control of the R4 data base. Having done that, TASKS.DONE in the control block is updated to indicate completion of the selected task. This process continues until all the tasks have been dispatched. When this is done, the triad exits the R4 frame by resetting its bit in the first field of TRIAD.TRACKER. If it is not the last triad to exit the frame, it kicks another triad to pick up R4. If, on the other hand, it is the last triad to exit the frame, it makes itself R4 responsible. To do so, the start time of the next R4 frame as indicated in the R4 control block is updated. This R4 start time is then compared to the current time to see if enough time (40 percent of the R4 frame time or Slop) is left to do the R3 and R1 processes. If not, the start of the next R4 frame is delayed to allow at least 40 percent of the R4 frame time to do lower priority processes. In any case, a procedure, SET.R4.RUPT.TIME, is called with the start time of the next R4 frame as its argument. This

procedure computes the correct interval with which to arm the interval timer.

The sequence of events is slightly different if a triad is commanded to retire. This command word is checked before dispatching a new task so that the response time to the command will be the minimum possible. If the triad is to retire and it is not the last one to exit the frame, it simply kicks another triad. However, if it is the last one out, it obviously cannot make itself R4 responsible. Therefore, it modifies the kick field of TRIAD.TRACKER (obviously if this is the last triad to quit, all other triads have already been kicked) so as to rekick another triad, thereby making sure that someone does start the next R4 frame. In any case, the retiring triad also resets its bit in TRIAD.VOLS indicating it is not available for work assignment.

This completes the task dispatch loop. The R4 control block is then updated in system memory and the R4 data base unlocked. The EXEC.LEVEL is restored to its original value and all previously halted timers restarted by calling a procedure RELEASE.R3.R1.TIMERS. The R1 and R3 user stack entries in the mapper are also restored to their original status.

The dispatcher then resumes the previously interrupted process.

The R3 and R1 dispatchers are very much the same with some obvious exceptions. The R4 responsible functions (I/O, etc.) are not performed. If it is the R3 dispatcher, the R1 timer is stopped at entry and restarted before exiting. Also the R1 user stack is write protected at entry and restored to its original status before exiting. Neither of these

functions is applicable to R1 dispatcher. Finally, the R3 and R1 dispatchers do not kick other triads.

The dispatchers use a number of Executive primitives. As mentioned earlier, all the timer related functions are collected together in FTMP.AED(TIMERS). The system bus related routines, those that access memory, real-time clock, and other control registers on the system bus are grouped together in FTMP. ASM(SERVICE). These are, however, not used to read error latches or clear them. These more efficient customized routines are collected under "error latch service routines." Finally, a number of miscellaneous primitives such as lock, unlock, kick, etc., are collected in the library FTMP.AED(GPROCS). All of these Executive primitives are described in Section 3.7.

3.5 System Configuration Control

The overall function of the FTMP system configuration controller is to maintain system integrity in the presence of hardware faults. It does this by detecting faults, identifying the faulty units, and replacing them with spares or gracefully degrading the system if no spares are available. Hard, as well as transient, faults are handled by the configuration controller. To assure a high level of system integrity at all times, spare units are periodically brought on-line. This includes processors, system memory units, all buses, and clock elements. To minimize reconfiguration time, spare processor and memory units are assigned to shadow active processor and memory triads. In addition, self-test programs are run continually on active elements to uncover any latent or lurking faults. Finally, a customized version of the controller communicates with the Fault Injection Software (FIS) resident in PDP-11 through the 1553 bus to facilitate the fault-injection and subsequent data collection process.

The program that performs all the functions outlined above is SCC (System Configuration Controller). This program runs as a privileged R1 rate group applications task. It calls various procedures to perform individual functions such as fault detection, hard failure analysis, deactivate failed units, assigning shadows, etc. Section 3.5.1 describes the overall configuration control structure by detailing the SCC procedure. The subsequent six sections describe in detail each major function such as fault detection, identification, etc.

3.5.1 SCC

This is the main system configuration control procedure. Its N-S diagram is shown in Figure 3-33. SCC, as well as most procedures called by SCC, have been structured to perform like state machines. That is, the function performed by the program on a given cycle depends on the state of the program. This has been done due to the fact that there is a great deal of continuity from one cycle to the next of these programs. For example, to identify a faulty bus may take up to four cycles of SCC. The actions taken by the identification program on a given cycle depend upon the past history. This knowledge about the past history of the system can be conveyed either through a data base, usually quite large, resident in the system memory or more briefly and efficiently through the state of the program.

The SCC program has four states as described in the following:

(1) Restart: The R4 dispatcher initializes the SCC program to this state upon system restart. SCC initializes its internal variables here and changes the program state to 'Normal Entry'.

(2) Normal Entry: As the name implies, this segment of the program is executed under normal circumstances. In this phase SCC calls an error detection program. The detection program reads 48 error latches and condenses the information into 4 words for the identification phase. If any errors are detected in this phase, the program state is changed to 'Identify'. Under normal circumstances there should not be any errors. In that case, routine configuration changes, as required by the current system status, are made. These are explained below.

```
READ PROGRAM STATE FROM SYSTEM MEMORY.
DONE = FALSE
WHILE NOT DONE
    GO TO PROGRAM STATE (ONE OF FOUR)
    END.OF.PROGRAM:
WRITE PROGRAM STATE TO SYSTEM MEMORY
```

RESTART: (STATE 1)

```
SET LATCH STATUS OF LRU'S NOT ENABLED ON R BUS TO 'FAILED'
INITIALIZE SUSPECT LIST & SWAPTIME
SET PROGRAM STATE = NORMAL.ENTRY.
GOTO END.OF.PROGRAM
```

Figure 3-33. SCC N-S diagram.

NORMAL.ENTRY: (STATE 2)

<table>
<tr><td colspan="6">PROCESS ERROR LATCHES</td></tr>
<tr><td colspan="6">ANY FAULTS DETECTED?</td></tr>
<tr><td>YES</td><td colspan="5">NO</td></tr>
<tr><td rowspan="11">SET PROM STATE = IDENTIFY
IDENTIFY STATE = 0
NO.OF.PASSES = 0</td><td colspan="5">3 OR MORE SPARE PROCS?</td></tr>
<tr><td>YES</td><td colspan="4">NO</td></tr>
<tr><td rowspan="9">SET
RECONF.STATE = 4
TO FORM A NEW
PROCESSOR TRIAD</td><td colspan="4">ANY IDLE SPARE PROC?</td></tr>
<tr><td>YES</td><td colspan="3">NO</td></tr>
<tr><td rowspan="8">SET
RECONF.STATE = 8
TO ASSIGN SHADOW
PROC</td><td colspan="3">ANY IDLE MEMORY?</td></tr>
<tr><td>YES</td><td colspan="2">NO</td></tr>
<tr><td rowspan="7">SET
RECONF.STATE = 11
TO ASSIGN SHADOW
MEMORY</td><td colspan="2">TIME TO CYCLE?</td></tr>
<tr><td>YES</td><td>NO</td></tr>
<tr><td>SET RECONF
STATE = 13
TO CYCLE
A SPARE</td><td>SET
DONE
= TRUE</td></tr>
<tr><td colspan="3">IS DONE = TRUE?</td></tr>
<tr><td colspan="2">YES</td><td colspan="2">NO</td></tr>
<tr><td colspan="2">NIL</td><td colspan="2">SET PROM STATE = RECONF</td></tr>
<tr><td colspan="6">GO TO END.OF.PROGRAM</td></tr>
</table>

Figure 3-33. SCC N-S diagram (cont.)

IDENTIFY: (STATE 3)

READ NO.OF.PASSES FROM SYSTEM MEMORY
INCREMENT NO.OF.PASSES
CONTINUE = TRUE

IS THIS FIRST TIME THRU?

YES: NIL

NO:

HAVE ALL RECONFIGURATION CMNDS BEEN EXECUTED?

YES:

PROCESS.ERROR.LATCHES()

NO:

CONTINUE = FALSE
DONE = TRUE

CONTINUE?

YES:

FAULTY.UNIT.FOUND = FALSE
HFA (HARD FAILURE ANALYSIS)

FAULTY UNIT IDENTIFIED?

YES:

PROGRAM STATE = RECONFIGURE
RECONF.STATE = 0

NO:

ASSIGN.DEMERITS()

NO.OF.PASSES > 4? OR NO RECONF CMNDS

YES:

TRANSIENT.FAULT.ANALYSIS()
PROGRAM STATE = RECONFIGURE

FAULTY UNIT FOUND?

YES:

RECONF.STATE = 1

NO:

IF NO RECONFCMND THEN ISSUE RECONF.-CMNDS AND SET RECONF.STATE = 7
ELSE SET RECONF.-STATE = 100

NO:

DONE = TRUE

NO: NIL

WRITE NO.OF.PASSES TO SYSTEM MEMORY.

GO TO END.OF.PROGRAM

Figure 3-33. SCC N-S diagram (cont.).

RECONFIGURE: (STATE 4)

IS RECONF STATE = 100?	
YES	NO
NIL	GOTO R.STATE (1 TO 14)
ALL DONE (STATE = 100)?	
YES	NO
CLEAR ALL ERROR LATCHES SET SWAP TIME PROM STATE = NORMAL ENTRY	NIL
SET REFRESH DISPLAY FLAG DONE = TRUE	

R.STATE 1:

TRANSFER 'LONE ACCUSER' DEMERITS TO PERMANENT FILE IF ANY ERROR LATCH HAS 2 OR MORE SUCH DEMERITS MARK IT AS FAULTY UNIT	
IS THE FAULTY UNIT MEMORY OR CLOCK?	
YES	NO
ISSUE RECONFIGURATION CMNDS TO DEACTIVATE ALL SUBUNITS OF THE TARGET LRU SET R.STATE = 2	ISSUE RECONFIGURATION CMND TO DEACTIVATE ONLY THE FAILED UNIT SET R.STATE = 3 OR 2 (BUS OR ANY OTHER FAILURE)

R.STATE 2:

IF DEACTIVATE CMNDS (1 TO 3) HAVE BEEN EXECUTED, THEN ALL DONE (SET R.STATE = 100)

R.STATE 3:

IF BUS DEACTIVATE CMND HAS BEEN EXECUTED, THEN ALL DONE (SET R.STATE = 100)

Figure 3-33. SCC N-S diagram (cont.).

R.STATE 4:

```
START 3 SPARE PROCESSORS TO FORM A NEW TRIAD
SET R.STATE = 5
RESET SWAP CYCLE
```

R.STATE 5:

```
ISSUE TRIAD ID TO PROCESSORS
SET R.STATE = 6
```

R.STATE 6:

```
ALL DONE (SET R.STATE = 100)
```

R.STATE 7:

```
IF ALL DIAGNOSTIC RECONFIGURATION CMNDS HAVE
BEEN EXECUTED, THEN SET R.STATE = 100
```

Figure 3-33. SCC N-S diagram (cont.)

R.STATE 8:

IS THERE A PROC TRIAD WITHOUT A SHADOW?	
YES	NO
ISSUE GOTO IDLE CMND TO TARGET TRIAD SET R.STATE = 9	SET PROM STATE = NORMAL ENTRY
RESET SWAP CYCLE	

R.STATE 9:

IS TARGET TRIAD IN IDLE MODE?	
YES	NO
DISABLE SPARE ON ALL P&T BUSES SET ITS R3 TO F ISSUE SYNCH CMND TO TARGET TRIAD SET R.STATE = 10	NIL

R.STATE 10:

SET R.STATE = 100 IF THE TARGET TRIAD HAS SYNCHRONIZED WITH THE SPARE

R.STATE 11:

ISSUE 'ASSIGN MEMORY SHADOW' CMND SET R.STATE = 12 RESET SWAP CYCLE

R.STATE 12:

SET R.STATE = 100 IF ASSIGN SHADOW CMND HAS BEEN EXECUTED

Figure 3-33. SCC N-S diagram (cont.).

R.STATE 13:

```
ISSUE SWAP CMND
SET R.STATE = 14
```

R.STATE 14:

```
SET R.STATE = 100 IF SWAP CMNDS HAVE BEEN EXECUTED
```

Figure 3-33. SCC N-S diagram (cont.).

First of all, system configuration tables are searched to see if there are three spare processors to form a new processor triad. This function is provided solely as a laboratory environment facility and would not normally be part of the flight version code. This facility enables the operator to retry failed processors. If three spares are available, the program state is changed to 'Reconfigure' and the state of reconfiguration program is also initialized appropriately.

If there are any spare processor or memory units then they are assigned to shadow active triads. This also entails a change in program state to Reconfigure and a different state within the reconfiguration segment. Forming a new processor triad or assigning shadows generally takes several passes of SCC. Once any of these operations has started, the SCC program stays in Reconfiguration state until the operation is complete. Error detection is suppressed during this phase. SCC returns to normal mode once the specified system configuration change has been completed.

If there are no more idle spares to be assigned as shadows, the system is put into a cycling mode. In this mode, shadow processors and memory units are swapped with active units and spare bus lines are interchanged with active bus lines. Of course, only one unit is swapped at a time. The swapping is done once every 10 seconds. When it is time to cycle, the SCC state is changed from normal mode to Reconfigure with the reconfiguration state initialized appropriately. Since cycling of spares happens only occasionally in relation to the SCC iteration rate, on most passes of SCC only the error detection program is run. During these low

workload cycles, self-test programs are run to uncover latent faults. If any faults are uncovered in this process, the SCC program state is changed to Reconfigure.

(3) Identify: This SCC program segment is first entered when a fault is detected by the detection phase of the SCC program. A hard failure analysis (HFA) program is called to process the four condensed error latches (one for each bus) produced by the detection phase. If the faulty unit can be identified based on this information, the SCC state is changed to Reconfigure. If, however, the information is not sufficient to isolate the faulty unit, a list of suspect units is saved in system memory and diagnostic reconfiguration commands issued to change system configuration in order to collect more data on the source of faults. The reconfiguration commands are executed in the R4 dispatcher prolog. If after four passes the hard failure analysis cannot identify the source of faults a transient fault analysis (TFA) program is called. TFA assigns demerits to all suspects and performs statistical analysis of data collected since the system was started up. If the data shows a unit to be above statistically significant threshold, then that unit is removed from service. The SCC program state for this purpose is changed to Reconfigure. If no significant trend is found, the program state is changed to Normal Entry and normal processing resumed.

The hard failure analysis can handle only one failure of a given type of unit at a given time. That is, a simultaneous failure of a processor and memory or a memory and P or T bus can be handled but not two processors, two memories, etc. Actually, any failure combination

that results in two out of three active buses of a given type to be in error would not be handled by HFA. If the system is still able to mask these errors and continue to perform normally, the transient fault analysis is used to identify the multiple sources of errors. TFA assigns demerits to all suspect units and eventually the faulty units would accumulate enough demerits to cross the identification threshold. It may be noted here that this situation would almost never occur during normal operation of the FTMP but can happen during system startup.

(4) Reconfigure: There are several ways of getting to this segment of SCC. Anytime the system needs to be reconfigured, except when the hard failure analysis is in progress, the SCC program state is changed to Reconfigure. There are 15 different states of this segment of SCC. These are explained as follows.

State 1 is entered from Identify phase when a faulty unit has been identified. In this state, appropriate reconfiguration commands are issued to deactivate failed units. As pointed out earlier, these commands are executed in the R4 dispatcher prolog by a procedure called CONCOM (configuration commands). The commands are stored in the system memory. The command words include the identity and type of failed unit such as processor 5 or R bus 2 etc. but do not specify the spare with which the failed unit may be replaced. The program that executes the configuration commands searches the system configuration table and then decides upon the optimum reconfiguration strategy. This program clears the command word in the system memory when it has finished executing the command. In any case, having issued the commands, the reconfiguration

state is changed to 2 or 3 depending on what commands were issued. In states 2 and 3, appropriate command words in system are checked to see if they have been cleared. At the completion of the commands, the reconfiguration state is changed to 100. Functions performed in state 100 will be explained shortly.

In addition to issuing deactivate commands, state 1 also handles the 'lying-fault syndrome'. That is, when a unit is deactivated based solely on the complaint of one other unit, the accuser is marked as a potential faulty unit. If in future the accuser should single-handedly complain about another unit then the accuser is deactivated first. For example, suppose processor 5 alone indicates through its error latches that processor 7 is making errors on the P or the T bus. As a result of this, processor 7 is deactivated but at the same time processor 5 is marked. If at some time in future processor 5's latches alone indicate that say, processor 2, is faulty, then processor 5 rather than 2 is deactivated. This strategy prevents a single unit from knocking out all others through the 'lying-fault syndrome'.

The next reconfiguration state is 4. This state is entered from normal SCC mode if there are enough spares to start a new processor triad. In this state, a program START.TRIAD is called. This program enables the three processors on appropriate P and T buses, enables their parent LRU's on R buses if they are not already enabled, issues zero triad id and run command to processors and updates system configuration tables. The reconfiguration state is then changed to 5. On the next pass of SCC, by which time the processors should have completed their initialization and synchronization phase, a valid triad id is issued to

the three processors. The state is changed to 6. On the next pass the state is changed from 6 to 100.

State 100 is the last state of reconfiguration segment. All reconfiguration program segments must eventually exit through this state. In this state, all the error latches are cleared, the spare swaptime is updated and the SCC program state is changed to Normal Entry.

The next state is 7. This state is entered from the identification phase after a transient fault has been observed and reconfiguration commands have been issued. The program stays in this state until the commands have been executed, at which time the state is changed to 100.

State 8 is entered from normal SCC mode if a spare processor has to be assigned to shadow a triad. The concept of shadowing a triad can be explained as follows. A shadow processor is in tight synchronism with the active members of the triad it is shadowing. It is assigned the same triad id as the active units. It listens to all commands on the buses. It receives IPC interrupts same as the active members. It participates in the bus poll. However, it is not enabled to transmit on the P or the T bus. Failure recovery for a triad with a shadow is speedier since it only involves swapping bus enables of the failed and the shadow processors. The triad does not have to wind down its task assignments and synchronize with a new processor member. In any case, in this state a program FIND.TRIAD is called. This program finds a processor triad that does not have a shadow and issues 'GOTO IDLE' command to the triad. The state is then changed to 9. On the next pass of SCC, in state 9, the target triad is checked to see if it has gone to idle process. If it has

a 'SYNCH' (synchronize with spare), command is issued to the target triad. The spare processor's control registers are initialized and configuration tables are updated. State is changed to 10. The program stays in state 10 until the 'SYNCH' command has been executed. Then the state is changed to 100.

The next reconfiguration state is 11. This is entered from normal SCC mode if a spare memory is available to be assigned as a shadow. A shadow memory has the same memory relocation (MRR) as the active members of the triad it is shadowing. However, its 'write only' bit is set in MRR while the RTC is disarmed. Therefore, the shadow memory responds to memory write requests but it does not respond to read requests. Failure recovery for a triad with a shadow is speedier since its contents are already in agreement with the active triad members. A shadow memory may or may not be enabled on the R bus depending on whether its associated processor or I/O port is or is not active. The 'write only' bit in MRR, in any case, inhibits the shadow unit from transmitting on the R bus notwithstanding the fact that it may be enabled on the R bus. A procedure 'MEM.SHADOW.CMND' is called from this state. This procedure issues 'Assign Shadow' commands to be executed by CONCOM in R4 dispatcher prolog. There are two active memory triads. The spares are evenly divided amongst the two triads. Initially, when there are several spares all units to be assigned to shadow one triad are refreshed simultaneously. If one triad exhausts all its shadow units while the other triad still has more than one shadow, the shadow units are transferred from one triad to the other so that each triad has at least one shadow assigned to it.

Having issued the command the state is changed to 12. The program stays in this state until shadow units have been refreshed. The program state is then changed to 100 where all error latches are cleared and SCC state changed to Normal Entry.

The next reconfiguration state is 13. This is entered from normal SCC mode if 10 seconds have elapsed since the last swap of active and shadow units. A procedure ISSUE.SWAP.CMND is called from this state. This procedure determines which two units should be exchanged next and issues appropriate commands to be executed by SWAP.LRUS in R4 dispatcher prolog. The commands are stored in the system memory. The details of these two procedures, that is, ISSUE.SWAP.CMND and SWAP.LRUS, will be described in a subsequent subsection. In any case, having issued the commands the state is changed to 14. The program stays in this state until the swap commands have been executed. The state is then changed to 100.

Section 3.5.2 describes the fault-injector provisions of this program.

3.5.2 FSCC

A customized version of SCC, called FSCC has been written to facilitate automatic fault-injection in LRU 3 of the FTMP and subsequent data collection.

In this version of SCC, provisions have been made to communicate to the Fault Injection Software (FIS) package, running in the PDP-11, through the 1553 bus. The FIS can command a fault to be injected into LRU 3 of the FTMP. The objective is to measure the FTMP response time to

this event. The response time is broken down into three phases, fault detection time, identification time, and recovery time. Since the FTMP does not know the time at which the fault is injected, the FTMP time base (the Real Time Clock) is sent to FIS every R4 frame. The value of the RT Clock at the time the fault is detected, identified, and recovered is also sent to FIS. The FIS can then compute the actual detection, identification, and recovery times. The identification and recovery times can be computed with an accuracy equal to the least count of the RT Clock which is 1/4 millisecond. The detection time, however, has an error equal to ± 1/2 the R4 frame time since the fault injection time is known only to that accuracy.

In order to repeatedly inject a fault and observe the results, an FSCC-FIS communication protocol has been devised to allow the repair of LRU 3 after each fault-injection event. This protocol works as follows. When FIS is ready to inject a fault, it sends a GET.READY command to FSCC via 1553. FSCC looks at this word in its normal mode. If it is true, the FSCC state is changed to Reconfigure and the reconfiguraton state is initialized to 13. Recall that in SCC state 13 corresponds to cycling spare units. In FSCC spares are not cycled. Instead in this state status of processor 3 and memory 3 is checked. If they are failed, they are repaired by changing their status in the system configuration tables. The reconfiguration state is changed to 100 so that on the subsequent FSCC pass the spare units viz. processor and memory 3 can be assigned to shadow active triads. If the units were not failed, the state is changed to 14. In this state, swap commands are issued to swap

processor and memory 3 with active members of their parent triads. The state is changed to 15. Also, a signal called ACK.GET.READY is sent to FIS acknowledging that GET.READY command has been received and acted upon by FSCC. FIS then clears GET.READY. FSCC waits in state 15 until swap commands have been executed. It then sends a READY word to FIS indicating that LRU 3 components have been repaired and are in the active state. The detect, identify and recovery times to be sent to FIS are cleared to zero and the state is changed to 100.

When the fault is identified, the identity of the faulty unit and the reason code are also sent by FSCC to FIS. FIS keeps track of any misdiagnosed faults.

The next subsection describes the detection phase of the configuration control program.

3.5.3 Fault Detection

3.5.3.1 Fault Detection Methodology

Fault detection in FTMP relies upon hardware majority voting and subsequent disagreement detection between the majority voter output and each of the three inputs. The disagreements are stored in hardware registers called error latches. Each LRU has four error latches, one for each type of bus, that is, poll (P), transmit (T), receive (R) and clock (C) bus. The 48 latches from 12 LRU's are read as simplex-source non-voted data over the system R bus. In the presence of faults on the R bus itself, the error latches read over the faulty R bus line can therefore be corrupted. An R bus line may be unusable either because the bus line

itself is faulty or because an LRU is actively transmitting on the line. In any case, those error latches received over a faulty R bus must be discarded before using any error latch information for fault detection. Therefore, any faulty R bus, if there is one, must be identified even before latches can be used to detect and identify any faults!

To overcome this chicken and egg problem, each error latch is examined for three reasonableness tests. The three tests are as follows. A good latch should not indicate errors on an inactive bus line or on more than one active bus line and finally the leading 11 bits of the latch should all be 1. The first of these three tests is not applied to C latches since they compare all five C bus lines to the majority output. If a latch fails any test, the R bus line on which it was read is marked suspect. In addition, the remaining R latches which do meet all three tests are checked to see if they indicate errors on any R bus line. This R bus line is also marked as suspect. Then all latches read over the suspect R bus lines are discarded.

The next phase of the detection process is to condense the remaining latches into four words, one for each bus type, for the identification phase. To do this, error latches are further filtered through several masks. First of all, error latches from failed units are discarded. That is, P and T latches of failed processors or P, R, T, and C latches of failed memory units are discarded. Next, each latch is filtered through a bus mask to ignore errors on certain buses. For example, C latches are filtered through a C bus mask to ignore errors reported on the inactive C bus. The remaining latches for each bus type are then

'or'ed together to form one error word for each bus. The four words then indicate errors on the 20 bus lines. If any error is indicated, a flag in set by the fault detection program indicating to SCC that a fault has been detected. SCC would then invoke the identification program.

If there is only one faulty unit in the system, only one bus line should be flagged by an error word. However, if multiple units have failed, multiple bus lines may be flagged by error words. The fault identification program has two components, hard failure analysis (HFA) and transient fault analysis (TFA). HFA can identify faulty units only if one unit of a given type such as a processor, memory or LRU is failed. Multiple simultaneous failures are handled by TFA. Therefore, fault detection program converts error latch information into two sets of error words one for HFA and the other for TFA. The HFA error words are cleared if there are multiple errors.

Finally, if a bus error is detected only by a single LRU, the identity of that LRU is stored. This information is used by SCC to handle the 'lying-fault-syndrome' as explained in Section 3.5.1.

3.5.3.2 Fault Detection Program and Data Base

Figure 3-34 shows an N-S diagram of the fault detection program. This procedure, called PROCESS.ERROR.LATCHES, is called either from Normal Entry or Identify mode of SCC. In either case, it returns four error words for Hard Failure Analysis and four error words for Transient Fault Analysis. In addition, a word indicating the number of buses in

PROCESS ERROR LATCHES

READ 48 ERROR LATCHES FROM 12 LRU'S PERFORM SOURCE CONGRUENCY EQUALIZATION ON LATCHES READ SYSTEM CONFIGURATION TABLES FROM SYSTEM MEMORY
IF ANY GOOD LATCH HAS A MISSING LEADING 1, AN ERROR ON AN INACTIVE BUS OR ERRORS ON MULTIPLE BUSES, MARK THE R BUS ON WHICH THIS LATCH IS ENABLED UNUSABLE IF ANY GOOD LATCH WITHOUT ANY OF THE ABOVE SYMPTOMS INDICATES ERROR ON AN R BUS MARK THAT BUS SUSPECT OR UNUSABLE
IF A GOOD LATCH THAT IS ENABLED ON A USABLE R BUS (NOT SUSPECT) INDICATES ERROR ON AN UNMASKED BUS, SET BIT CORRESPONDING TO THAT BUS LINE IN THE ARRAY LATCH AND TR.LATCH
IF ANY WORD IN ARRAY LATCH INDICATES ERRORS ON MULTIPLE BUSES SET THAT WORD TO ZERO (LATCH ARRAY IS USED BY HFA, TR.LATCH BY TFA)

Figure 3-34. Fault detection N-S diagram.

error is also returned. This word is examined by SCC to decide if identify program should be invoked. Other details of the detection program are as described in the previous section.

The structure of various data base elements used by this program is shown in Figures 3-35 through 3-40. Figure 3-35 shows the ACTIVE.BUS array. This is a four-word array. There is one word for each bus type. The 5 least significant bits (LSB's) of each word show the active bus lines. A 1 in one of these bit positions corresponds to an active bus line. Figure 3-36 shows the SPARE.BUS array. The structure of this array is similar to that of ACTIVE.BUS. A 1 in one of the least significant 5 bits corresponds to a spare bus line. Figure 3-37 shows the BUS.MASK array which is also similar in structure to the previous two arrays. A 1 in one of the 5 LSB's indicates that errors on that bus line should be masked or ignored by the fault detection program.

The next two arrays, LATCH and TR.LATCH, are both 4 word arrays and have structure as shown in Figure 3-38. There is one word for each bus type. A 1 in one of the 5 LSB's corresponds to an error on that bus line. LATCH and TR.LATCH are the error word arrays produced by the fault detection program for HFA and TFA, respectively. Figure 3-39 shows the structure of the ERR.LATCH array which is 48 words long. This array contains the raw error latch readings from the 12 LRU's. They are organized in 12 groups, each group containing P, R, T, and C latches from an LRU. The 11 most significant bits of each entry should normally be all 1. A 1 on one of the 5 LSB's indicates error on that bus line. Figure 3-40 shows the LATCH.STATUS array. This is a four-word array, one word

ACTIVE.BUS: 4 WORD ARRAY

THERE IS ONE WORD FOR EACH BUS TYPE (P, R, T & C).
EACH WORD SHOWS THE ACTIVE BUS LINES (5 LEAST SIGNIFIACANT BITS)
FOR A BUS TYPE. A '1' IN A BIT POSITION CORRESPONDS TO AN
ACTIVE BUS LINE.

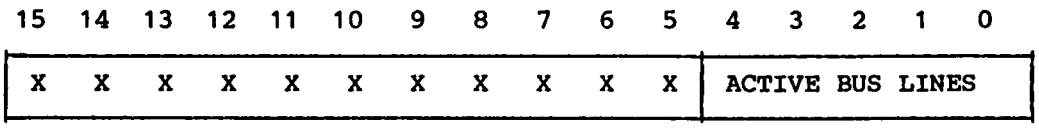


Figure 3-35. ACTIVE.BUS.

SPARE.BUS: 4 WORD ARRAY

THERE IS ONE WORD FOR EACH TYPE OF BUS (P, R, T & C).
EACH WORD SHOWS THE SPARE BUS LINES (5 LEAST SIGNIFIACANT BITS)
FOR A BUS TYPE. A '1' IN A BIT POSITION CORRESPONDS TO A SPARE BUS LINE.

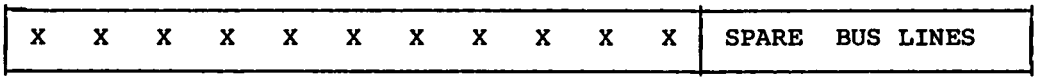


Figure 3-36. SPARE.BUS

BUS.MASK: 4 WORD ARRAY

THERE IS ONE WORD FOR EACH TYPE OF BUS (P, R, T & C).
EACH WORD SHOWS THE BUS LINES ON WHICH ERRORS SHOULD BE IGNORED (5 LEAST
SIGNIFICANT BITS). A '1' IN A BIT POSITION IMPLIES THAT ERRORS ON THAT
BUS LINE SHOULD BE IGNORED WHEN PROCESSING ERROR LATCHES.

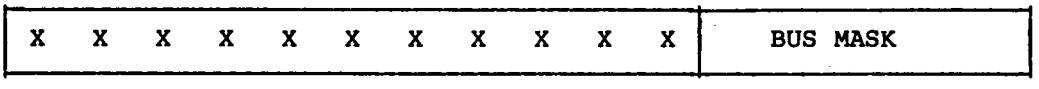


Figure 3-37. BUS.MASK.

LATCH: 4 WORD ARRAY

EACH WORD SHOWS ERRORS ON BUS LINES (5 LEAST SIGNIFIACANT BITS) FOR P, R, T AND C BUSES. A '1' IN A BIT POSITION CORRESPONDS TO AN ERROR ON THAT BUS LINE.

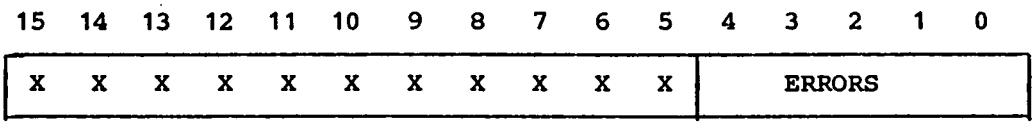


Figure 3-38. LATCH and TR.LATCH.

ERR.LATCH: 48 WORD ARRAY

THIS ARRAY CONTAINS THE RAW ERROR LATCH READINGS FROM 12 LRU'S. THESE ARE ORGANIZED IN 12 GROUPS, EACH GROUP CONTAINING 4 LATCHES FROM AN LRU.

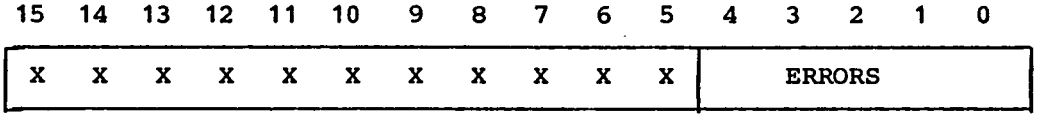


Figure 3-39. ERR.LATCH.

LATCH.STATUS: 4 WORD ARRAY

THIS ARRAY CONTAINS THE STATUS OF 48 ERROR LATCHES. THERE IS ONE BIT FOR EACH ERROR LATCH. A '1' INDICATES THAT THE ERROR LATCH IS FAILED. THE 12 LEAST SIGNIFICANT BITS IN EACH WORD CORRESPOND TO THE STATUS OF LATCHES IN 12 LRU'S FOR A GIVEN BUS. THE 4 WORDS CORRESPOND TO P, R, T AND C BUSES RESPECTIVELY.

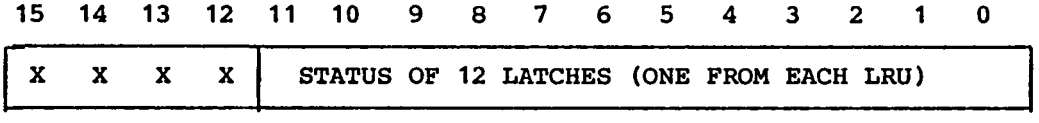


Figure 3-40. LATCH.STATUS.

for each bus type. The 12 LSB's in each word correspond to the status of the 12 latches in 12 LRU's for a given bus type. The four words correspond to P, R, T, and C buses respectively. A 1 in a bit position indicates that the latch from that LRU is to be ignored.

3.5.4 Fault Identification

The error latches in the FTMP indicate only the bus lines which disagree with the majority. The identity of the source transmitting on that bus line at the time the disagreement is detected is not recorded. Since up to 3 LRU's can be enabled on a given P or T bus and up to 4 on an R bus the identity of the faulty unit cannot be ascertained simply with the help of the error latch information. The fault identification process is therefore slightly more involved than might appear at first.

Sometimes the fault is such that the source of the fault is obvious. At other times, however, a number of units can be attributed the fault. In that case it is necessary to reconfigure the system and observe the results to resolve the ambiguity in the source of the fault. This process may have to be repeated several times depending upon the number of units enabled on the bus in error and the source of fault. A fault that does not persist through this process, which may take several hundred milliseconds, can not be identified simply by associating errors on a bus with units enabled on that bus. Two fault detection strategies, therefore, have been provided, viz., Hard Failure Analysis (HFA) and Transient Fault Analysis (TFA). HFA is always tried first if the fault disappears while the HFA is in progress, the TFA is then invoked. The next two sections describe these two programs.

3.5.4.1 Hard Failure Analysis

The input to this procedure is the 4 word error array, called LATCH, produced by the fault detection program. The array LATCH indicates which of the P, R, T, and C bus lines are in error. Depending on the type of fault, anywhere from 1 to all 4 buses could be in error. Each of these four cases are handled by four different programs.

Before describing the four programs, it is interesting to note the failure signatures of different components in the FTMP. The broadest failure signature is painted by an LRU failure due to one of the common mode failures. For example, if the power supply or the clock in an LRU fails, all the buses on which that LRU is enabled would show errors. If the processor and clock of that LRU are active, errors would appear on the associated P, R, T, and C bus lines. This type of failure is the easiest to locate simply by determining which LRU is enabled on the four buses in error. Another multi-symptom failure signature is one caused by a processor failure. Most processor failures result in errors on the P and the T bus. Initial system configuration has been arranged such that each processor is assigned to a unique P and T bus combination. LRU and processor failures, therefore, can be identified in a single pass of HFA. Bus and memory failures can appear on single or multiple buses depending on the type of fault. A memory array failure, such as a stuck-at-0 or stuck-at-1 bit, would show up only as an R bus error. However, a fault in the memory controller such that it does not respond to read requests would cause error latches from that LRU to fail reasonableness checks in addition to causing errors on the R bus. If a single

bus line is faulty, errors would be limited only to that bus. However, a whole bus set can be shorted together in an LRU causing errors on one bus line of each type of bus.

The four programs that handle the four classes of errors are described in the following.

3.5.4.1.1 Single Fault Analysis

This procedure is called by HFA when there is an error only on one bus. An N-S diagram of this program is shown in Figure 3-41. This program, like most other SCC programs, is state driven. The state number of this program is the number of suspect elements. On each pass, a list of suspect elements is made. This list includes the suspect bus, that is, the bus in error, and all LRU's enabled on that bus. This number can vary from 1 for a C bus to 4 for an R bus. In any case, on the first pass, the suspect list is saved in the system memory and reconfiguration commands are issued to distribute suspect LRU's on different buses. The program state is initialized to number of suspects and saved for the next pass. On the next pass the suspect LRU is identified if the error moved with the LRU to a different bus. If the error stayed on the same bus, the list of suspects is narrowed. The remaining suspect LRU's are once again distributed on different buses by reconfiguring the buses. This process continues until the suspect LRU or the suspect bus is confirmed to be the faulty unit. If the faulty unit is neither an LRU nor a bus but a combination of them, that is, an LRU unable to transmit on a particular bus line, the error would disappear as soon as the faulty

ANY ERRORS ON THIS PASS?	
YES	NO
FIND LRU'S ENABLED ON THE 'BAD' BUS. IF THIS IS NOT THE FIRST PASS AND ERROR THIS TIME IS ON A DIFFERENT BUS FROM PREVIOUS TIME THEN SET HFA STATE = 0 (START OVER AGIAN). GO TO HFA STATE.	IF HFA STATE IS 2 (ONLY 1 LRU & 1 BUS ARE SUSPECTS) THEN INTERSECTION OF THIS LRU & BUS IS FAULTY. CONFIRM BY REENABLING SUSPECT LRU ON SUSPECT BUS.
STATE 0: (FIRST PASS) SUSPECTS ARE THE BUS AND ALL THE LRU'S ENABLED ON IT ISSUE APPROPRIATE RECONFIGURATION COMMANDS DEPENDING ON THE BUS IN ERROR P & T BUSES: ROTATE PROCESSOR TRIADS ON P/T BUS R BUS: ROTATE MEMORY TRIADS ON R BUS (IF 2 TRIADS THEN ROTATE 1ST LEFT, ROTATE 2ND RIGHT) C BUS: EXCHANGE OSCILLATOR ON THE C BUS WITH ONE ON THE SPARE SET HFA STATE = NO OF SUSPECTS (2 TO 4)	
STATE 2: (TWO SUSPECTS: ONE BUS, ONE LRU) COMPARE THE SUSPECT BUS LINE ON THIS PASS TO THE ONE FROM THE PREVIOUS PASS. IF THE ERROR STAYED WITH THE SAME BUS THEN THE BUS IS FAULTY. IF THE ERROR MOVED WITH THE SUSPECT LRU THEN THE LRU (PROCESSOR, MEMORY OR OSCILLATOR) IS FAULTY. IF NEITHER OF THESE IS TRUE THEN START OVER AGAIN. SET FAULTY.UNIT & HFA.STATE ACCORDINGLY.	
STATE 3: (SUSPECTS: ONE BUS, 2 LRU'S) IF ERROR 'MOVED LEFT' THEN THE FIRST SUSPECT LRU IS THE FAULTY UNIT. IF ERROR 'MOVED RIGHT' THEN THE SECOND SUSPECT LRU IS THE FAULTY UNIT. IF ERROR STAYED ON THE SAME BUS THEN THE BUS IS FAULTY.	
STATE 4 AND 5: (SUSPECTS: ONE BUS, 3 or 4 LRU'S) IF ERROR 'MOVED LEFT' THEN THE FIRST SUSPECT LRU IS THE FAULTY UNIT. IF ERROR 'MOVED RIGHT' THEN THE SECOND SUSPECT LRU IS THE FAULTY UNIT. IF ERROR STAYED ON THE SAME BUS THEN THE LIST OF SUSPECTS NARROWED TO 2 OR 3 UNITS: 1 BUS & 1 OR 2 LRU'S	
WRITE THE LIST OF SUSPECTS TO SYSTEM MEMORY.	

Figure 3-41. Single fault analysis.

combination is broken. This fault is identified as a 'weak intersection' of the LRU and the bus. It is confirmed by enabling the suspect LRU back on the suspect bus.

If at any time during this process of localizing the source of the fault, the error on the bus disappears completely or the error appears on an unexpected bus, the hard failure analysis is terminated and the transient fault analysis is invoked by the Identify program.

3.5.4.1.2 Double Fault Analysis

This procedure is called by HFA if two buses are in error. A list of suspect LRU's is formed by determining which units are enabled on the two suspect buses. A program called FIND.COMMON.UNITS is called for this purpose. The two error words are passed as arguments to this procedure. If only a single common unit is found, the suspect LRU is identified in a single pass. If several LRU's are found, then they are distributed on different buses. On the next pass a new list of suspects is formed and compared to the old list to determine any LRU's that are common to both. Finally, if no single unit can be attributed errors on the two suspect buses, then the bus lines are compared to see if a bus set may be faulty.

3.5.4.1.3 Triple Fault Analysis

This procedure is very similar to the previous one. System configuration tables are searched to determine if a single LRU or bus set can be the faulty unit. In case of ambiguity, the system is reconfigured and suspect LRU's on the two passes are compared to reveal any common LRU.

3.5.4.1.4 Quad Fault Analysis

This procedure is same as the previous one except that it handles four suspect buses.

3.5.4.2 Transient Fault Analysis

The transient fault analysis is based on the concept of 'fault index'. Each element in the FTMP, that is, processor, memory, bus line, and clock element, is assigned a fault index. This index is the fraction of all relevant unresolved faults in which the element was a suspect. As such, the index would vary from 0 to 1. Not all faults are relevant to all elements. For example, an error on the P bus incriminates processors and P bus lines but not memory, clock, or other bus types. An unresolved fault is one that was not attributed to a single unit by HFA.

In the full-up system configuration (3 processor triads, 2 memory triads, 5 bus lines of each type and 10 clock elements), the fault-index of good processors would hover around 0.33. The numbers for memory, bus, and clock are 0.33, 0.20, and 0.10, respectively. The fault-index of faulty units, on the other hand, would tend towards 1.0 in the presence of a single fault source of a kind. The fault indices of all elements are initialized to their expected good values. Every time a transient fault occurs, these fault indices are revised as explained in the following.

The new fault index for those elements which are suspect in the present incidence is computed as follows.

$$\text{NEW FI} = \text{OLD FI}.(1 - X) + X.$$

X is a weight assigned to the latest fault incidence. The choice of X depends upon the required response time in isolating the faulty unit and the probability of false alarm or false identification. X is chosen to be 1/8. For those units which are not implicated by the present fault incidence the fault-index is reduced by the following formula.

$$\text{NEW FI} = (\text{OLD FI} - X) \cdot \frac{1}{1-X}.$$

A fault isolation threshold is assigned to each type of element. If the fault-index of a unit crosses this threshold, that unit is declared to be faulty. At the same time the fault-index of all other relevant units is reset to their initial value. The thresholds for processor, memory, bus, and clock are 0.7, 0.7, 0.6 and 0.7, respectively.

There are two programs which perform the transient fault analysis. These are ASSIGN.DEMERITS and TFA. ASSIGN.DEMERITS is called from Identify phase of SCC on each pass. Its input is the 4 word error array TR.LATCH prepared by the fault detection phase. It examines the system configuration tables and assigns a demerit count to each suspect unit. If the faulty unit is found by HFA, the demerits are thrown away and a new demerit count started with the next occurrence of a fault. If HFA cannot locate the faulty unit, a program called TFA is called. This program transfers the temporary demerits assigned by ASSIGN.DEMERIT to a permanent form by recomputing the fault-index of all the implicated units. The fault-index of units not implicated is also recomputed as described previously. A unit is not implicated if it is active but not

enabled on the suspect bus. Spare units, therefore, are not assigned this good merit badge. Two separate fault-indices are kept for processors, one for errors on the P bus and a second for errors on the T bus. A processor is deactivated if either of its two fault-indices crosses the threshold. If a processor is thrown out due to its P fault-index, then only the P fault-index of other units is reset.

3.5.4.3 SCC, HFA, and TFA Data Base

The data sets used by SCC, HFA, and TFA, in addition to those described in Section 3.5.3.2, are as follows.

All variables to be described in this section are used from one iteration to the next and they are all saved in the system memory. The states of the main SCC program, HFA, and Reconfigure are saved in a 3 word array SCC.STATE. This array is initialized after the system restarts. An 8 word array, SUSPECT.LIST, is used to store the number of suspects, the ID's of up to 4 suspect LRU's, suspect bus type and bus line and the number of buses reporting errors. A 4 word array, EL.DEMERIT, is used to assign temporary demerits to 'lone accusers'. The structure of this array is same as that of EL.STATUS described earlier. A 12 word array EL.ERR is used to hold permanent count of 'lone accusers'. There is one word for each LRU. Each word has 4 4-bit nibbles, one each for P, R, T and C latches. Two arrays, SUBUNIT.ERR (12 words) and BUS.ERR (5 words) are used by ASSIGIN.DEMERIT to assign temporary transient fault demerits to various FTMP elements. Each element of these arrays has four 4-bit fields. The four fields of SUBUNIT.ERR elements

are used for processor (P bus errors), memory, processor (T bus errors), and clock demerit counts. The four fields of BUS.ERR are similarly used for P, R, T, and C bus demerit counts. The permanent demerits or fault-indices are in two arrays SUBUNIT.INDEX (48 words) and BUS.INDEX (20 words). The first array, SUBUNIT.INDEX, is organized as a 12 × 4 word array. The first 12 words of the array are fault-indices for the 12 processors (P bus errors), the next 12 for memory, the next 12 for processors (T bus errors), and the last 12 for the clocks. The second array, BUS.INDEX, is organized as a 5 × 4 word array. The first 5 words are fault-indices of the 5 P bus lines. The subsequent 5 word groups are fault-indices of the R, T, and C buses respectively. There are two other words used by SCC. These are NO.PASSES, which is the number of HFA iterations performed to isolate a fault, and FAULTY.LATCH which is set to TRUE if any error latch demerits are assigned to lone accusers.

3.5.5 Spare Cycling

All spare elements in the FTMP, processors, memories, buses, clocks, are periodically activated to discover any latent faults as pointed out earlier in Section 3.5.1. Under normal circumstances, that is, in the absence of any failures, a procedure ISSUE.SWAP.CMND is called from SCC every 10 seconds. This procedure determines which spare unit should be brought on-line and issues appropriate swap commands which are executed by the R4 dispatcher prolog.

On each pass only one spare unit is swapped with an active unit. There are two swap cycles, major and minor. During a major cycle, one

spare bus of each type is made active. A major cycle has four minor cycles. During each minor cycle, each processor and memory triad goes through one swap and one of the four bus types also is swapped once.

There is a word in the system memory, RESET.SWAP.CYCLE, which is set to TRUE by SCC after any change in the system configuration due to a failure. At that point, the major cycle is reset and cycling begins at the top of the cycle. A cycle begins with a processor triad, goes through all processor triads with shadows, then goes through all memory triads with shadows and the P bus. This minor cycle is repeated three more times except that the R, T, and C buses are swapped on the subsequent three iterations. The four minor cycles constitute one major cycle. The details of the program are as follows.

The procedure ISSUE.SWAP.CMND is a state driven program and has 3 states, 1, 2, and 3 which correspond to processor, memory, and bus swap. In state 1 a procedure SWAP.PROC is called with the ID of the target triad as an argument. If the target triad has a shadow, a command is issued by SWAP.PROC to swap the shadow with an active member. SWAP.PROC also sets a variable DONE to TRUE. The target triad ID is incremented. If all processor triads with shadows have gone through a swap, the program state is changed to 2. If SWAP.PROC returns the variable DONE set to FALSE indicating the target triad did not have a shadow, SWAP.PROC is called once again with the next triad ID as its argument. This ensures that a swap command is issued every time ISSUE.SWAP.CMND is called.

State 2 actions are similar to that of state 1. A procedure SWAP.MEM with MRR of the target triad as its argument is called. This

procedure issues a command to swap one of the shadows of the target triad with one of its active member. When all memory triads with shadows have gone through a swap, the program state is changed to 3.

In state 3 a procedure SWAP.BUS with the type of bus to be swapped (P, R, T, or C) as its argument is called. It issues an appropriate swap command and the program state is changed back to 1. The following describes the three swap procedures.

3.5.5.1 SWAP.PROC

This procedure is called with the ID of the target triad as its argument. It examines a 3-word array NUM.SH.PT in the system memory to see if the target triad has a shadow. If it does, the LRU ID of the shadow processor is obtained from a 3-word array SHADOW.PT. The ID of the active unit with which the shadow is to be swapped is obtained from a 9-word array ACTIVE.PT. This array holds LRU ID's of the active members of the three processor triads. A 3-word pointer array APNTR.PT has pointers which point to active processors to be swapped next from each triad. This ensures that each active member in turn is rotated into the spare state. A swap command is formed once the active and the shadow processor ID's have been determined. This command is stored in system memory to be executed by the R4 dispatcher prolog. The pointer in the APNTR.PT array is updated and a boolean DONE is set to true. If the target triad dots not have a shadow, no action is taken.

The data sets used by this procedure such as ACTIVE.PT, SHADOW.PT etc. are initialized by a procedure SETUP.PROCESSORS. This procedure is

called by ISSUE.SWAP.CMND if RESET.SWAP.CYCLE is true. SETUP.PROCESSORS examines the system configuration tables and determines active members of all processor triads, their shadows, if any, and stores this information in the data sets used here. If the system configuration changes due to any failures the swap data sets are reinitialized by SETUP.PROCESSORS with the new system configuration information. Reformatting of the system configuration information for swapping significantly reduces computations by the swap procedures.

3.5.5.2 SWAP.MEM

This procedure is called with the ID of the target memory triad as its argument. The swap data sets used by this procedure are initialized by SETUP.MEMORY.

First of all, NUM.SH.MT array is examined to see if the target triad has a shadow. A memory triad, unlike a processor triad, may have several shadows. The shadow and the active units to be swapped are obtained from the arrays SHADOW.MT and ACTIVE.MT, respectively. SPNTR.MT and APNTR.MT arrays hold pointers to the next shadow and active units, respectively. A swap command is formed once the ID's of target units have been determined. No action is taken if the target triad does not have a shadow. DONE is set to TRUE if a swap command is issued.

3.5.5.3 SWAP.BUS

This procedure is called with the bus type to be swapped as its argument. The swap data sets used by this procedure are initialized by SETUP.BUS.

First of all, NUM.SP.BUS array is examined to see if the target bus type has a spare. APNT.BUS AND SPNTR.BUS arrays point to the next active and spare buses to be swapped. The two target bus numbers are obtained by using these pointers as indices into ACTIVE.BUS and SPARE.BUS arrays. An appropriate swap command is issued and DONE is set to TRUE. No action is taken if there is no spare bus.

3.5.6 System Reconfiguration

All the system reconfiguration commands are executed by the R4 dispatcher prolog. The commands are inserted in the system memory by SCC. There are four types of commands as follows.

(1) Unit Deactivate Commands: These commands are issued to deactivate a failed active unit. There are four such command words, one each for processor, memory, clock, and bus. The structure of each of these words is shown in Figure 3-42. These words are located in the system memory in the CC.COMMAND array.

(2) Diagnostic Reconfiguration Commands: These commands are issued to change bus assignments of processors, memories, and clocks. They are contained in two words which are also part of the CC.COMMAND array. The structure of these two command words is shown in Figure 3-42.

(3) Swap Commands: These commands are issued to swap active and shadow units. There are four such command words contained in the SWAP.COMMANDS array. Their structure is shown in Figure 3-43.

THE FIRST WORD IN THE COMMAND WORD ARRAY IS THE 'DEACTIVATE PROCESSOR' COMMAND. THE COMMAND IS NULL IF THE WORD IS ZERO. FOR A NON-ZERO WORD VARIOUS FIELDS OF THE WORD ARE INTERPRETED AS SHOWN BELOW:
THE LRU NO. OF THE PROCESSOR TO BE DECOMMISSIONED IS IN BITS 8-11.
THE LOW ORDER BYTE CONTAINS THE PROCESSOR STATUS AND ITS PARENT TRIAD ID
THIS ENTRY IS THE SAME AS THE ENTRY FOR THIS LRU IN TRIAD ID TABLE.

WORD 0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	LRU ID				0	0	PROC STAT		0	TRIAD ID		

SECOND WORD IN THE COMMAND WORD ARRAY IS THE 'DEACTIVATE MEMORY' COMMAND. BITS 8-11 ARE THE LRU NO OF THE MEMORY MODULE.
THE LOW ORDER BYTE IS THE SAME AS THE ENTRY IN MRR TABLE FOR THIS LRU.

WORD 1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	LRU ID				MEMORY STATUS		RT ENBL	RELOCATION CONST				

THIRD WORD IN THE COMMAND WORD ARRAY IS THE 'DEACTIVATE OSCILLATOR' COMMAND. BITS 8-11 ARE THE LRU NO OF THE OSCILLATOR.
THE LOW ORDER BYTE IS THE SAME AS THE ENTRY IN TC TABLE FOR THIS LRU.

WORD 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	LRU ID				OSC STATUS		0	C BUS ENABLES				

FOURTH WORD IN THE COMMAND WORD ARRAY IS THE 'DEACTIVATE BUS' COMMAND. BITS 0-3 ARE THE BUS IDENTIFICATION (0, 1, 2 & 3 FOR P, R, T AND C RESPECTIVELY. 4 IMPLIES THE WHOLE BUS SET, I.E., P,R,T & C)
BITS 8-12 ARE THE BUS LINE IDENTIFICATION (I.E. BUS LINE 1,2,4,8 OR "10"

WORD 3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	BUS LINE ID					0	0	0	0	0	BUS TYPE		

Figure 3-42. Deactivate and Rotate commands.

FIFTH WORD IN THE COMMAND WORD ARRAY IS THE 'ROTATE LEFT' COMMAND. BITS 0-3 ARE THE BUS IDENTIFICATION (0, 1, 2 & 3 FOR P, R, T AND C RESPECTIVELY) AND BITS 8-11 ARE THE LRU NO. THAT IS TO BE ROTATED.

WORD 4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	LRU ID				0	0	0	0	0	BUS TYPE		

SIXTH WORD IN THE COMMAND WORD ARRAY IS THE 'ROTATE RIGHT' COMMAND. BITS 0-3 ARE THE BUS IDENTIFICATION (0, 1, 2 & 3 FOR P, R, T AND C RESPECTIVELY) AND BITS 8-11 ARE THE LRU NO. THAT IS TO BE ROTATED.

WORD 5

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	LRU ID				0	0	0	0	0	BUS TYPE		

Figure 3-42. Deactivate and Rotate commands (cont.)

(4) Assign Shadow Command: Finally, this command is issued to assign a shadow memory. It is in the system memory word MEMORY.COMMAND. Its structure is shown in Figure 3-44.

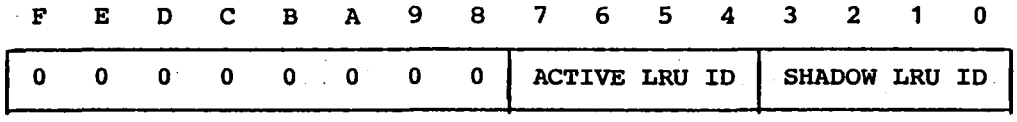
The R4 dispatcher calls a procedure RECONF to execute these reconfiguration commands. The procedure RECONF examines each command word in the system memory and calls appropriate procedures to execute the commands. There are nineteen such procedures. These are described in the following subsections.

3.5.6.1 Deactivate Processor

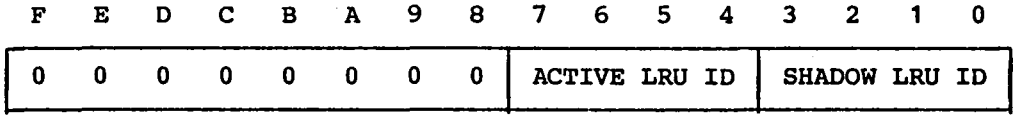
The object of this procedure is to replace the failed processor whose ID is passed in the command word by a suitable shadow or spare processor. If neither of these options is feasible, the parent triad of the failed processor is retired and the remaining two good processors are thrown into the spare pool. An N-S diagram of the procedure is shown in Figure 3-45.

The failed processor can be replaced by a spare in one pass if that spare is assigned to shadow the parent triad. In this case, the failed processor in reset and disabled on P and T buses and the shadow processor is enabled on the same P and T buses. Also, if the parent LRU of the shadow processor is not enabled on an R bus, the LRU is assigned to one of the active R buses. The R bus is chosen such that there are no more than three LRU's already enabled on it.

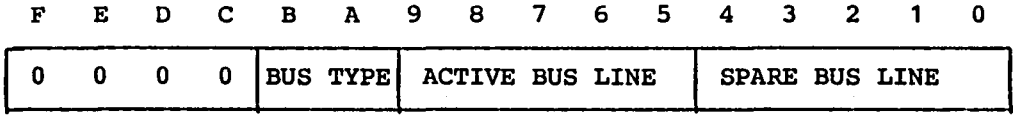
If the target triad is not being shadowed by a processor but either an idle spare or a spare that is shadowing another triad is



Swap Processor Command



Swap Memory Command



Swap Bus Command

Figure 3-43. Swap commands.

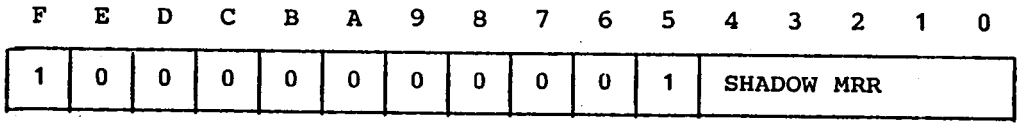


Figure 3-44. Assign shadow command.

<table>
<tr><td colspan="4">IS PROGRAM STATE = 0? (FIRST TIME THRU?)</td></tr>
<tr><td colspan="2">YES</td><td colspan="2">NO</td></tr>
<tr><td colspan="2">IS TARGET PROCESSOR A SPARE?</td><td colspan="2">IS TARGET TRIAD IN IDLE PROCESS YET?</td></tr>
<tr><td>YES</td><td>NO</td><td colspan="2">YES</td><td>NO</td></tr>
<tr><td rowspan="2">MARK PROCESSOR FAILED IN TRIAD ID TABLE
HOG BUS.
DISABLE PROC ON ALL P & T BUSES.
ISSUE 0 TRIAD ID AND RESET TARGET PROC

RELEASE BUS.
SET PRGM STATE =4 (ALL DONE)</td><td rowspan="2">IF TRIAD HAS A SHADOW ENABLE IT ON P, T BUSES AND SET STATE = 4. IF NOT, ISSUE GOTO IDLE CMND TO TARGET TRIAD (IF MULTIPLE TRIADS) ELSE SET SYNCH.LAST.-TRIAD TO TRUE AND STATE = 2</td><td colspan="2">READ TRIAD ID TABLE FROM SYSTEM MEMORY.
SEARCH THE TABLE TO FIND A SPARE PROCESSOR.</td><td rowspan="3">N

I

L</td></tr>
<tr><td colspan="2">IS A SPARE PROCESSOR AVAILABLE?</td></tr>
<tr><td>YES</td><td>NO</td></tr>
<tr><td colspan="2"></td><td>OBTAIN P & T ENABLES OF TARGET PROCESSOR.
HOG BUS.
DISABLE TARGET PROC ON P & T BUSES.</td><td>HOG BUS.
DISABLE PROC ON P & T BUSES
ISSUE 0 TRIAD ID AND RESET PROCESSOR.
RELEASE BUS.
MARK TRIAD FAILED IN TRIAD ID TABLE
UPDATE P & T ENABLE TABLES FOR THIS LRU

SET PRGM STATE = 4 (ALL DONE)</td><td></td></tr>
<tr><td colspan="2"></td><td colspan="2">ISSUE 0 TRIAD ID & RESET TARGET PROC. RESTART SPARE PROC. ENABLE SPARE ON SAME P & T BUSES AS TARGET PROC. ISSUE 'SYNCH' COMMAND TO TARGET TRIAD THRU AN IPC INTERRUPT. RELEASE BUS. UPDATE TRIAD ID TABLE FOR SPARE AND TARGET PROCESSORS. UPDATE P & T ASSIGNMENT TABLES FOR THE 2 PROCESSORS. SET PROGRAM STATE = 4 (ALL DONE)</td><td></td></tr>
</table>

Figure 3-45. Deactivate processor.

available, the target triad is issued a 'GOTO.IDLE' command. On subsequent passes of this procedure this command word is checked to see if the triad has retired, that is, gone to Idle mode. At that time, two different actions are taken depending upon whether a spare processor is available or not. If a spare is available, it is enabled on appropriate P, R, and T buses, and its control register R3 is initialized to F. The target triad is sent an IPC interrupt to pend RESTART process and a SYNCH command is also issued to the target triad to synchronize with the spare processor. The SYNCH command word is checked on subsequent passes to see if the triad has synchronized with the spare. At that point, the procedure returns an All Done code to RECONF signifying completion of 'Deactivate Processor' command. If a spare is not available, the target triad is dismembered by resetting the two good members of the triad and disabling them on the P and T buses.

A special case occurs when only one processor triad is left in the system. In this case, the GOTO.IDLE command is not issued. Instead a flag, SYNCH.LAST.TRIAD, in the cache memory is set to TRUE and the SYNCH command is also issued at the same time. The flag is checked by each triad in the idle process. If the flag is set, the Restart process is activated from the idle process. In the Restart process, the triad synchronizes with the new processor and bootstraps R4 dispatcher directly.

In all cases, the P and T Latch status of the failed and the newly activated units is adjusted to reflect their changed status in the system.

3.5.6.2 Deactivate Memory

The object of this procedure is to replace the failed memory whose ID is passed in the deactivate command word with a suitable shadow or spare unit. An N-S diagram of this procedure is shown in Figure 3-46.

If the parent triad of the failed module is being shadowed by a spare memory, the triad can be repaired in one pass. The failed unit is disabled on the R bus and issued an MRR of 1E. The shadow is enabled on the R bus and its MRR is changed to clear the write only bit and arm the Real Time Clock, if necessary. The R latch status of the failed and the shadow units is changed too. In case there is no shadow memory but a spare is available, all of the above mentioned actions are taken. In addition, one 256 word page of the spare memory is refreshed by reading that page of the triad and writing it back while holding the bus. On the subsequent 63 passes of this procedure the remaining 63 pages of the spare memory are refreshed. A word, REFRESH.ADR, in the system memory is used to store the current refresh address.

3.5.6.3 Deactivate Clock

This procedure is called to replace a failed clock element by a spare. Its N-S diagram is shown in Figure 3-47. The bad clock element or oscillator is disabled on the C bus by writing into the BGU registers of its parent LRU. A spare oscillator, if one is available, is enabled on the C bus. Their C latch status is adjusted accordingly.

<table>
<tr><td colspan="5">IS PROGRAM STATE = 0? (FIRST TIME THRU?)</td></tr>
<tr><td colspan="3">YES</td><td colspan="2">NO</td></tr>
<tr><td colspan="3">IS THE TARGET MEMORY A SPARE?</td><td colspan="2" rowspan="2">OBTAIN LAST REFRESH ADDRESS FROM SYSTEM MEMORY.

HOG BUS.
READ ONE SYSTEM MEMORY PAGE STARTING AT REFRESH ADR.
WRITE THAT PAGE BACK.
RELEASE BUS.</td></tr>
<tr><td>YES</td><td colspan="2">NO</td></tr>
<tr><td rowspan="4">HOG BUS.
DISABLE MEMORY ON ALL R BUSES
REISSUE AN MRR "F" TO TARGET MEMORY.
MARK MEMORY FAILED IN MRR TABLE & UPDATE MRR ENTRY IN THE TABLE.
RELEASE BUS.
SET PRGM STATE =4 (ALL DONE)</td><td colspan="2">READ MRR TABLE FROM SYSTEM MEMORY.
SEARCH THE TABLE TO FIND A SPARE OR SHADOW MEMORY MODULE.
IF A SHADOW IS AVLBL ISSUE IT CORRECT MRR AND SET STATE = 4.</td><td colspan="2" rowspan="2"></td></tr>
<tr><td colspan="2">IS A SPARE MEMORY AVAILABLE?</td></tr>
<tr><td colspan="2">AVAILABLE?</td><td colspan="2">ALL 16K REFRESHED?</td></tr>
<tr><td>YES</td><td>NO</td><td>YES</td><td>NO</td></tr>
<tr><td>OBTAIN R ENABLES & MRR OF TARGET.
HOG BUS.
DISABLE TARGET MEMORY ON ALL R BUSES. ISSUE MRR = "F" TO TARGET MEMORY.
ENABLE SPARE UNIT ON THE SAME R BUS AS THE TARGET.
RELOCATE SPARE TO SAME ADDRESS SPACE AS THE TARGET AND ARM ITS REAL TIME CLOCK
IF THE TARGET MEMORY WAS ARMED UPDATE PRT, R ASSIGN & MRR TABLES FOR THE TARGET & THE SPARE UNIT. RELEASE BUS. INITIALIZE REFRESH ADR. SET PRGM STATE = 2.</td><td>HOG BUS.
DISBALE TARGET UNIT ON ALL R BUSES.
ISSUE MRR = "F" TO THE TARGET. UPDATE PRT, R ASSIGNMENT AND MRR TABLES.
RELEASE BUS.

SET PRGM STATE = 4 (ALL DONE)</td><td>SET PRGM STATE = 4 (ALL DONE)</td><td>UPDATE REFRESH ADR IN SYSTEM MEMORY.</td></tr>
</table>

Figure 3-46. Deactivate memory.

<table>
<tr><td colspan="3">IS THE TARGET OSCILLATOR A SPARE?</td></tr>
<tr><td>YES</td><td colspan="2">NO</td></tr>
<tr><td rowspan="3">UPDATE OSC STATUS IN TC TABLE.
UPDATE C ASSIGN TABLE.

SET PRGM STATE = 4
(ALL DONE)</td><td colspan="2">READ TC TABLE FROM SYSTEM MEMORY.
SEARCH THE TABLE FOR A SPARE OSC.</td></tr>
<tr><td colspan="2">IS A SPARE OSC AVAILABLE?</td></tr>
<tr><td>YES</td><td>NO</td></tr>
<tr><td>OBTAIN C BUS ENABLES OF TARGET OSC.
HOG BUS.
DISABLE TARGET OSC ON ALL C BUSES.
ENABLE SPARE OSC ON THE C BUS OF TARGET.
UPDATE TC & C ASSIGN TABLES
RELEASE BUS.
SET PRGM STATE = 4
(ALL DONE)</td><td>HOG BUS.
DISABLE TARGET OSC ON ALL C BUSES.
UPDATE TC TABLE & C ASSIGN TABLE.
RELEASE BUS.

SET PRGM STATE = 4
(ALL DONE)</td></tr>
</table>

Figure 3-47. Deactivate oscillator.

3.5.6.4 Deactivate P Bus

The object of this procedure is to replace a failed P bus line with a spare P bus line. Its N-S diagram is shown in Figure 3-48.

If a spare bus is available, the new 3-out-of-5 bus select code is computed. There are sixteen such codes, 0 to 15, 9 of which are legal. A 16 word array in the system memory, called BUS.SELECT.CODES, lists which 3 buses are selected by each of these sixteen codes. The structure of this array is shown in Figure 3-49. The new select code is obtained from this table. All LRU's are issued this new code by writing into their select registers. Then each processor that is enabled on the failed P bus is moved to the spare P bus by writing into its BGU registers. The P bus mask is changed to ignore errors on the failed P bus but accept errors on the spare P bus.

If no spare bus is available, only the bus mask is changed to ignore errors on the failed bus.

3.5.6.5 Deactivate R Bus

The object of this procedure is to replace a failed R bus line with a spare R bus line. Its N-S diagram is shown in Figure 3-50.

This procedure is very similar to the P bus procedure. It, however, performs one additional function. It computes the new 2-bit R bus simplex code for each LRU. To do this a 16 word array, called SIMPLEX.CODES, is read from the system memory. The structure of this array is shown in Figure 3-51.

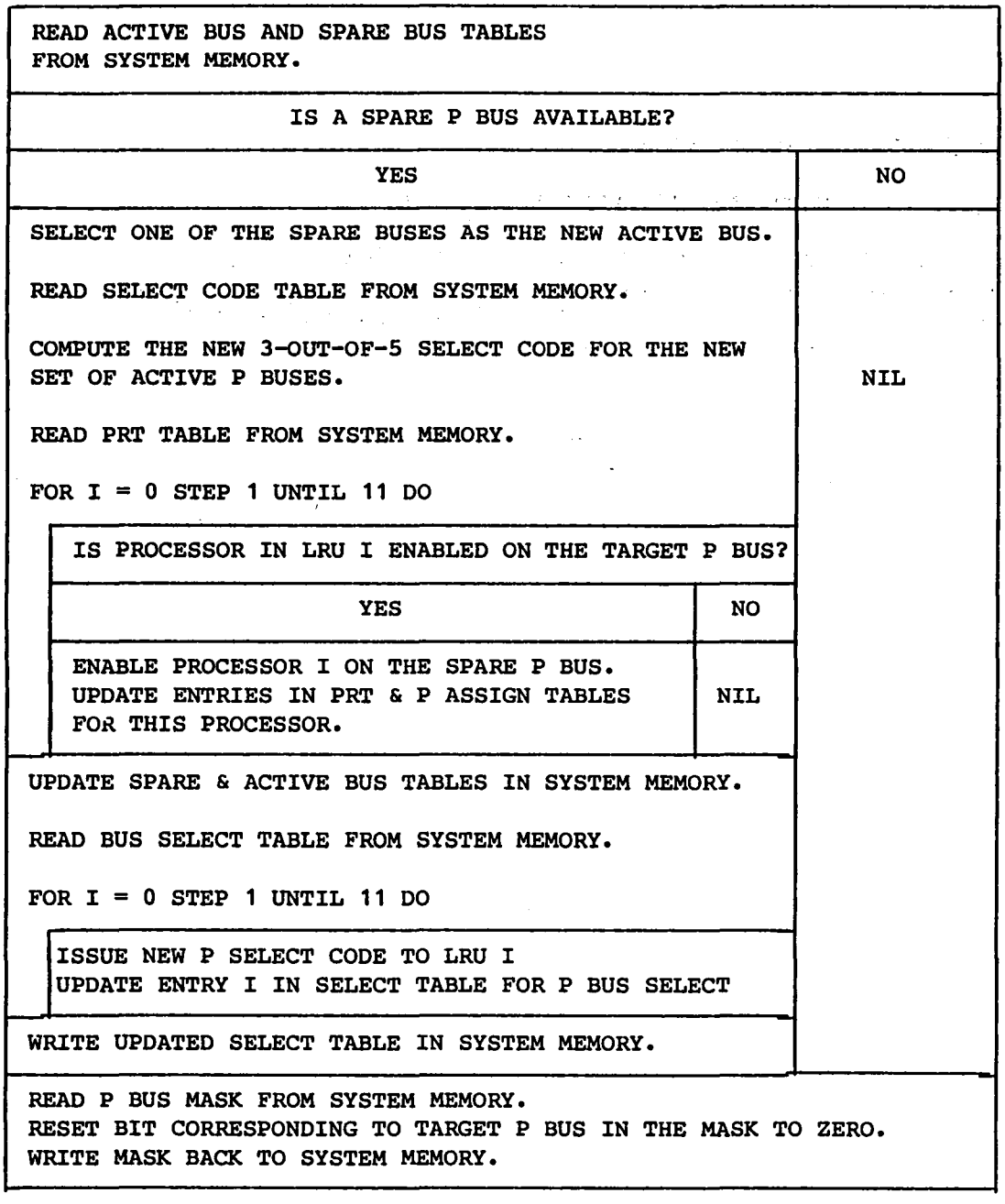


Figure 3-48. Deactivate P Bus.

BUS.SELECT.CODES: 16 WORD ARRAY

ITH WORD HAS THE 3 BUSES SELECTED BY CODE I.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	X	X	X	X	X	X	X	3 BUSES SELECTED BY I				

Figure 3-49. Bus select codes.

READ ACTIVE BUS AND SPARE BUS TABLES
FROM SYSTEM MEMORY.

IS A SPARE R BUS AVAILABLE?

YES	NO
SELECT ONE OF THE SPARE BUSES AS THE NEW ACTIVE BUS. READ SELECT CODE TABLE FROM SYSTEM MEMORY. COMPUTE THE NEW 3-OUT-OF-5 SELECT CODE FOR THE NEW SET OF ACTIVE R BUSES. READ SIMPLEX CODE TABLE FROM SYSTEM MEMORY. DETERMINE THE 2-BIT SIMPLEX CODE CORRESPONDING TO SELECTION OF THE NEW R BUS FROM THE ACTIVE R BUS SET. READ PRT TABLE AND BASE TRIAD TABLE FROM MEMORY. FOR I = 0 STEP 1 UNTIL 11 DO	NIL

IS MEMORY IN LRU I ENABLED ON THE TARGET R BUS?	
YES	NO
ENABLE MEMORY I ON THE SPARE R BUS. UPDATE ENTRIES IN PRT & R ASSIGN TABLES FOR THIS MEMORY. IF MEMORY I IS A MEMBER OF THE BASE TRIAD THEN UPDATE R ASSIGNMENT IN THAT TABLE ALSO	NIL

UPDATE SPARE, ACTIVE & BASE TRIAD TABLES IN SYSTEM MEMORY.
READ BUS SELECT TABLE FROM SYSTEM MEMORY.

FOR I = 0 STEP 1 UNTIL 11 DO

ISSUE NEW R SELECT CODE TO LRU I UPDATE ENTRY I IN SELECT TABLE FOR R BUS SELECT

WRITE UPDATED SELECT TABLE IN SYSTEM MEMORY.

READ R BUS MASK FROM SYSTEM MEMORY.
RESET BIT CORRESPONDING TO TARGET R BUS IN THE MASK TO ZERO.
WRITE MASK BACK TO SYSTEM MEMORY.

Figure 3-50. Deactivate R Bus.

SIMPLEX.CODES: 16 WORD ARRAY

EACH WORD HAS 5 2-BIT ENTRIES. THE ITH WORD CORRESPONDS TO BUS SELECT CODE I. EACH 2 BIT ENTRY IN THIS WORD IS THE SIMPLEX CODE FOR THAT BUS.

15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
BUS 1	BUS 2	BUS 3	BUS 4	BUS 5	X X	X X	X X

Figure 3-51. Simplex codes.

3.5.6.6 Deactivate T Bus

An N-S diagram of this procedure which is very similar to the P bus procedure is shown in Figure 3-52. It performs one additional function. The new T select code is issued to the BGU's as well as the LRU's.

3.5.6.7 Deactivate C Bus

An N-S diagram of this procedure is shown in Figure 3-53. This procedure is similar to the P bus procedure with the following exceptions. The new C bus select codes are computed such that active clock elements listen to the three clock elements other than themself. Also if no spare bus is available, all LRU's are issued a new select code to listen to the three good C buses.

3.5.6.8 Rotate Triad on P Bus

This procedure performs one of the diagnostic reconfigurations. It reassigns the P buses of the three processor members of the target triad. The target triad is the parent triad of the processor whose LRU ID is passed in the command word. The three processors are 'rotated' either 'left' or 'right' on the P bus as requested in the command word. A 'rotate left' involves enabling each processor on a higher numbered P bus while a 'rotate right' involves enabling each processor on a lower numbered P bus. For example, let us say processors 6, 7, and 8 are members of a triad and are enabled on P buses 3, 1, and 4. After a left rotation, the new bus assignments will be 4, 3, and 1 while it will be 1, 4, and 3 after a right rotation. The details of the procedure are as follows.

READ ACTIVE BUS AND SPARE BUS TABLES
FROM SYSTEM MEMORY.

IS A SPARE T BUS AVAILABLE?

YES	NO
SELECT ONE OF THE SPARE BUSES AS THE NEW ACTIVE BUS. READ SELECT CODE TABLE FROM SYSTEM MEMORY. COMPUTE THE NEW 3-OUT-OF-5 SELECT CODE FOR THE NEW SET OF ACTIVE T BUSES. READ PRT & TC TABLES FROM SYSTEM MEMORY. FOR I = 0 STEP 1 UNTIL 11 DO	NIL

IS PROCESSOR IN LRU I ENABLED ON THE TARGET T BUS?	
YES	NO
ENABLE PROCESSOR I ON THE SPARE T BUS. UPDATE ENTRIES IN PRT, TC AND T ASSIGN TABLES FOR THIS PROCESSOR.	NIL

UPDATE SPARE & ACTIVE BUS TABLES IN SYSTEM MEMORY.

READ BUS SELECT TABLE FROM SYSTEM MEMORY.

FOR I = 0 STEP 1 UNTIL 11 DO

ISSUE NEW T SELECT CODE TO LRU I
ISSUE NEW T SELECT CODE TO BGU'S IN LRU I
UPDATE ENTRY I IN SELECT TABLE FOR T BUS SELECT

WRITE UPDATED SELECT TABLE IN SYSTEM MEMORY.

READ T BUS MASK FROM SYSTEM MEMORY.
RESET BIT CORRESPONDING TO TARGET T BUS IN THE MASK TO ZERO.
WRITE MASK BACK TO SYSTEM MEMORY.

Figure 3-52. Deactivate T bus.

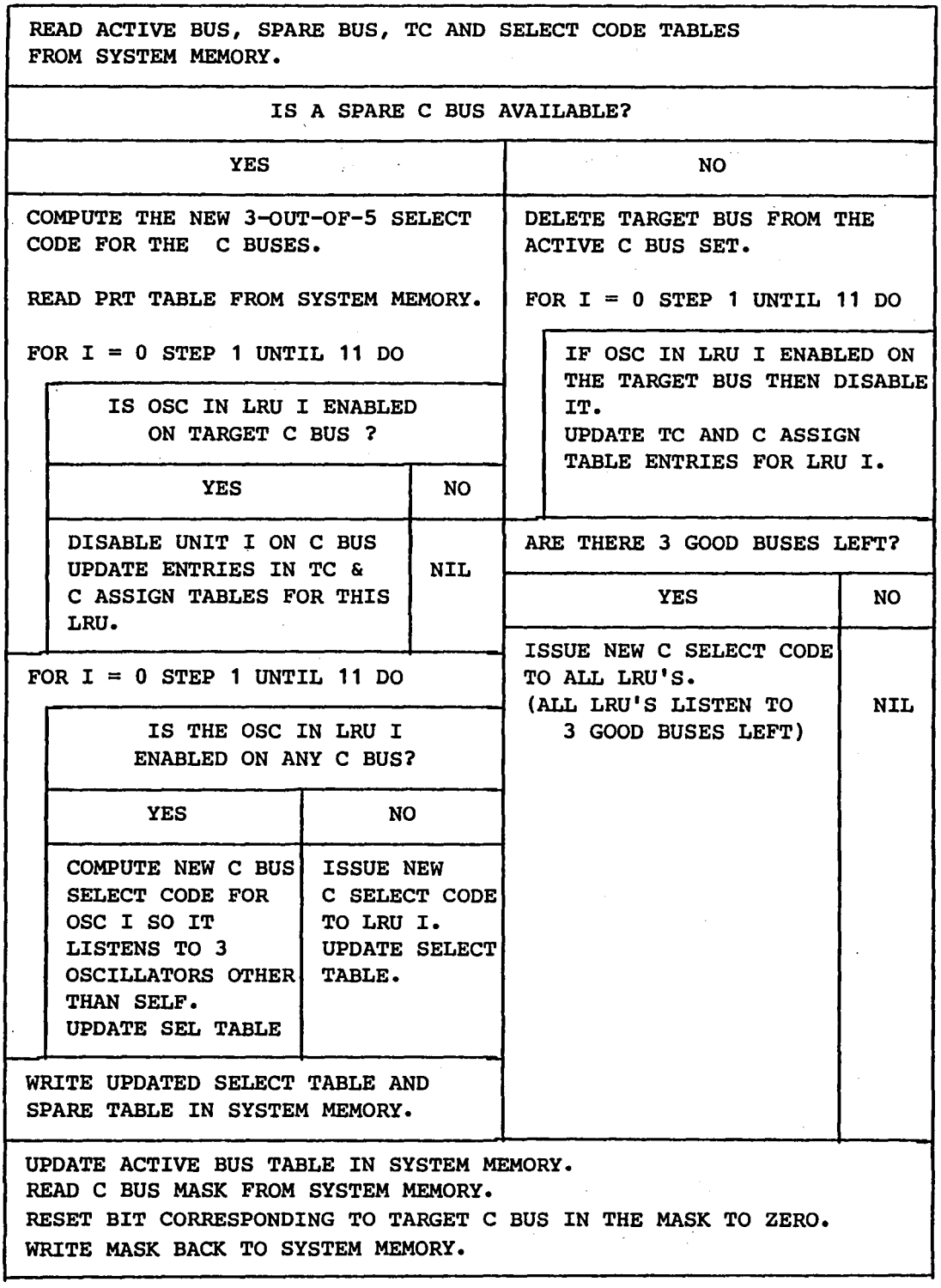


Figure 3-53. Deactivate C bus.

First of all, two accompanying members of the processor whose LRU ID is passed in the command word are determined by searching the TRIAD.ID.TABLE. The three processors are then arranged in an ascending or descending order of P bus assignments depending upon whether it is a left or a right rotate command. The new P bus assignments are then determined simply by rotating the array containing the three LRU ID's. To actually reassign the bus enables, each processor is first assigned on two P buses, the old and the new. Next, each processor is assigned to the new P bus only.

3.5.6.9 Rotate Triad on T Bus

This procedure performs one of the diagnostic reconfigurations. It rotates a triad on the T bus. The rotation is to the left or right as requested in the command word. This procedure is identical to the Rotate P Bus procedure except that processors are directly moved from their old T buses to the new ones without the intermediate step of enabling them on both the old and the new.

3.5.6.10 Rotate Memory

This procedure performs one of the diagnostic reconfigurations. It rotates the target memory triad on the R bus to the left or right as requested in the command word if the memory whose ID is passed in the command word is an active triad member. If the target memory is a spare or a shadow, it is simply assigned to a higher or a lower numbered active R bus depending upon whether the commanded direction is left or right.

If another spare or shadow is enabled on the new R bus it is moved to the old R bus to keep the maximum number of LRU's on any R bus to 4.

3.5.6.11 Exchange Oscillators

This procedure performs one of the diagnostic reconfigurations. It exchanges the C bus assignment of the oscillator whose ID is passed in the command word with that of a good oscillator clock element. The details of the procedure are as follows.

The target clock is disabled on the C bus. Call this bus the target bus. One of the other three clock elements of the clock quad, a good clock, is then enabled on the target bus in addition to its own C bus or the good C bus. Finally, the target clock is enabled on the good C bus and the good clock is enabled only on the target C bus.

3.5.6.12 Shadow Memory

This procedure executes the Assign Shadow Memory command. It refreshes one 256-word page of target memory unit(s) on each pass. All 16K words are refreshed in 64 passes of this program.

3.5.6.13 Swap Processor

This procedure swaps a shadow and an active processor. The ID's of the two units are passed in the swap command word. The active processor is disabled on the P and T buses and the shadow processor is assigned to those same P and T buses. In addition, if the shadow is not enabled on any R bus, an active R bus with 3 or less LRU's already

enabled on it is found and the shadow is enabled on it. The system status tables are updated as is the latch status for the two processors.

3.5.6.14 Swap Memory

This procedure swaps a shadow and an active memory. The ID's of the two units are passed in the swap command word. If the shadow is not enabled on any R bus it is assigned to one as described in the previous procedure. Then the MRR registers, R bus assignments and simplex codes of the shadow and the active unit are swapped.

3.5.6.15 Swap Oscillator

This procedure swaps the C bus assignments of the two clock elements whose ID's are passed in the command word.

3.5.6.16 Swap P Bus

This procedure swaps an active P bus with a spare P bus. The two bus line numbers are passed in the command word. This procedure is almost identical to the Deactivate P bus procedure. The only difference is that the spare bus identity is passed as an argument and it does not have to search system status tables to find a spare bus.

3.5.6.17 Swap R Bus

This procedure swaps an active R bus with a spare R bus. The two bus numbers are passed in the command word. This procedure also is almost identical to Deactivate R Bus with the exception noted above.

3.5.6.18 Swap T Bus

This procedure swaps an active T bus with a spare T bus. The two bus numbers are passed in the command word. It is almost identical to Deactivate T Bus.

3.5.6.19 Swap C Bus

This procedure swaps an active C bus with a spare C bus. The two bus numbers are passed in the command word. It is almost identical to Deactivate C Bus.

3.6 Input/Output

All the input/output in the FTMP is performed through the I/O ports on the MIL-STD-1553 bus. There are 11 I/O ports in the FTMP. These are connected to seven 1553 emulators over fully duplex 1553 buses as shown in Figure 1-1. Four of the seven buses are tied to two I/O ports each, while the remaining three are connected to one port each. One of the seven 1553 emulators is connected to an HP terminal through an RS232 interface. This terminal is used for FTMP status display and as an operator's monitor station or a console. The other six 1553 emulators, each of which can directly access PDP-11's memory (DMA into PDP-11), form the interface between the FTMP and a network of I/O nodes simulated in the PDP-11. This network is used to exchange information between the Boeing 707 simulation running in dual VAX 11/780 computers and the FTMP. Finally, one of the 1553 emulators is also used to communicate between the Fault Injector Software (FIS), which is resident in the PDP-11, and the FTMP System Configuration Controller (SCC).

The basic I/O routines in the FTMP, then, perform three functions, viz., 1) display and monitor I/O, 2) aircraft sensor and actuator I/O and 3) FIS-SCC communication. The first of these is performed by a procedure called TTY. The other two functions are performed by a set of six procedures: R4.IN, R4.OUT, R3.IN, R3.OUT, R1.IN and R1.OUT. These six procedures perform either the input or the output function and they run at R4, R3 or R1 iteration rates as indicated by their names. The information exchanged between the aircraft simulation and the applications tasks in the FTMP is divided into these three rate groups. Slowly changing items such as navigation data are updated at the lowest frequency while critical items such as the aircraft attitude information are updated at the highest frequency. The FIS-SCC communication is done at the R4 frequency. In any case, all I/O is done by the R4 dispatcher prolog.

There are actually only two procedures that perform the 1553 I/O transactions. One of these is TTY. The other is SENSOR.IO which is called with appropriate arguments from R4.IN, R4.OUT, R3.IN, R3.OUT, R1.IN and R1.OUT. These two procedures are described in the next two subsections.

3.6.1 TTY

This procedure is called by the R4 dispatcher in the dispatcher prolog. It performs a single 1553 transaction on each pass. This transaction involves either reading input typed on the FTMP console or sending display data to the console. The maximum number of characters transferred in either direction in a single pass is 64 (32 words). The maximum data rate to the console is 480 characters/second.

This procedure uses one of two I/O ports that are connected to the console via the 1553 emulator. If the primary I/O port fails or if the 1553 I/O transaction cannot be completed using the primary port, then all further I/O is performed through the secondary port. If this port fails too, then the primary port is checked once again in case it has been repaired through operator action. The designation primary and secondary is arbitrary.

The 1553 emulator which interfaces with the console has input and output buffers which are used to hold data being sent from and to the console, respectively. Before commencing any transaction for the console, the TTY program sends a "status" request to the 1553 emulator. The 1553 microcode sends back three words in response to the status request. The first of these words is the "terminal" or console status indicating any error flags set by the previous RS232 transaction to the terminal. The second word is the number of bytes, say M, waiting in the input buffer of the emulator. The third word is the number of bytes available, say N, in the output buffer of the emulator.

If there is any input from the console, that is, if M is not zero then M bytes or 64, whichever is smaller, are read by TTY from the emulator. This is accomplished by doing several 1553 bus transactions. These will be explained shortly. The input obtained from the terminal is stored in a circular input buffer in the FTMP system memory. The Display program empties the input buffer and takes appropriate action as commanded by the input data. If there is no input from the terminal, that is if M is zero,then the FTMP output buffer is checked for any data to be

transmitted to the console. This data, if any, is sent to the emulator provided the emulator output buffer has any room left in it. In any case, only a maximum of 64 characters are transmitted in one pass of this program. The following describes a typical 1553 bus transaction.

The I/O port status register is read by doing a simplex source read (SREAD) on the system bus. The status word thus obtained is written back to system memory using a source congruency write (SWRITE). The status word is then read into cache using a normal read (READ). The ready bit in the status word is checked to see if the I/O port is ready for a 1553 transaction. The I/O port status is checked a maximum of 50 times if the port is not ready. If the port is still not ready, the second I/O port is tried. Assuming the port is ready, a "Transmit RS232 Status" command is sent to the emulator by writing the command register of the port. The command register is written into by doing a high write (HWRITE). The ready bit in the I/O port is then checked, using the procedure described above, for the port to complete the 1553 transaction. At the completion of this transaction the requested status words should be in the port FIFO (First In First Out) buffer. These are read by doing a non-incrementing simplex read (SNREAD). The three status words are then processed through the source congruency filter, that is, they are written to system memory using SWRITE and read back using READ.

Assuming there is input, say M bytes, available from the console, a "transmit M bytes" command is sent to the emulator using the procedure described earlier. When the port completes the transaction, the FIFO is read to obtain the terminal input and then written into the FTMP circular input buffer.

If there is no input but output waiting to be sent to the terminal, the FIFO is loaded with the data, say N bytes, from the FTMP circular output buffer and a "receive N bytes" command word is written into the command register. Since this is the last 1553 transaction in this procedure, it does not wait for the port to complete the transaction before returning control to the R4 dispatcher.

3.6.2 SENSOR.IO

This I/O driver is similar to TTY. However, it does not have the RS232 protocol. It performs a single 1553 transaction, either input or output, as dictated by the arguments passed to it. It also uses a pair of I/O ports. However, these two ports are distinct from the pair used by TTY. SENSOR.IO also switches from one port to the other if the port it is using is marked failed or if it cannot complete a 1553 transaction due to errors.

The single transaction performed by this procedure is either transmission of 32 words from the FTMP to 1553 or vice versa. The buffer area in the FTMP and the direction of transfer are passed to it as arguments. The data contained in the buffer is aircraft sensors, navigation and guidance information, actuator commands or data being exchanged by FIS and SCC.

3.7 Executive Primitives

Various parts of the FTMP Executive described in the previous five sections of this chapter repeatedly use some basic functions. These are

collectively called the Executive primitives. They can be divided into four functional categories: System Bus Service Routines, Error Latch Service Routines, Timer Routines, and Miscellaneous Primitives. These are described in the following four subsections.

3.7.1 System Bus Service Routines

These programs are used to read/write all the devices (except error latches) available on the system bus. These include the system memory, the I/O port registers and FIFO buffer, the BGU registers, the IPC registers, the bus in mux selectors, the CPU control registers and the real-time clock. Since these registers and the system memory are accessed quite frequently, it was necessary to write very efficient programs to perform the required functions. The programs are, therefore, written in the CAPS assembly language and the source code for them is contained in the library FTMP.ASM (SERVICE). The assembled output is contained in FTMP.LIST (SERVICE). A total of 16 different functions are provided. However, there are only three main routines that actually perform the system bus transactions. The 16 functions are provided by 16 entry points into the core routines. Each entry point defines a procedure. The functions and arguments of each of the 16 procedures will be described next.

(1) READ (SYS.ADR, CACHE.ADR, NUM). This procedure transfers NUM number of words from system memory address SYS.ADR to cache address CACHE.ADR. It is assumed here that a page boundary either in the system memory or in the cache will not be crossed. The bus transaction is

terminated by the hardware if such a condition should occur. Arguments are not checked by the software for this abnormal condition. NUM cannot be zero. If it is, the processor would hang up. Software does not check for this condition either.

(2) WRITE (SYS.ADR, CACHE.ADR, NUM). This procedure is the same as READ except, of course, the direction of transfer is reversed. All the READ comments are equally applicable here.

(3) HREAD (SYS.ADR, CACHE.ADR, NUM). This procedure reads the high address space on the system bus. The high system bus address space includes all the devices mentioned above except the system memory. The 19-bit system bus address is formed by adding "70000" to the 16-bit address supplied in the argument SYS.ADR.

(4) HWRITE (SYS.ADR, CACHE.ADR, NUM). This procedure writes devices in the high system bus address space. HREAD and HWRITE are typically used to access Real-Time Clock, BGU registers, Bus in-mux Select Registers, IPC registers and CPU control registers.

(5) NREAD (SYS.ADR, CACHE.ADR, NUM). This procedure performs a non-incrementing read, that is, it reads NUM number of words from a single high system memory address "70000" + SYS.ADR into cache starting at location CACHE.ADR.

(6) NWRITE (SYS.ADR, CACHE ADR, NUM). This procedure performs a non-incrementing write. NREAD and NWRITE are typically used to access the I/O page FIFO buffer which is 32 words deep but has a single system bus address.

(7) SREAD (SYS.ADR, CACHE.ADR, NUM, SIMP.CODE). This procedure performs a non-voted simplex high read. NUM number of words are transferred from "70000" + SYS.ADR to cache address CACHE.ADR. However, unlike the other read functions described, only a simplex source is chosen. All other reads listen to three sources on three buses and the data obtained is the voted majority output. Here, the data is read over a single bus which is determined by the simplex code SIMP.CODE. Error detection circuitry (R bus error latches) is suppressed during SREAD. SREAD is typically used to read simplex sources such as I/O port registers and error latches.

(8) SWRITE (SYS.ADR, CACHE.ADR, NUM). This procedure, source congruency write, is used to transfer NUM number of words from cache to system memory while suppressing errors on the T bus. This procedure is normally used to write that data to system memory which has been read using simplex source read routines.

(9) SLREAD (SYS.ADR, CACHE.ADR, NUM, SIMP.CODE). This procedure performs a non-voted simplex read in the low address space. It is typically used by self-test programs to read a single system memory module.

(10) SNREAD (SYS.ADR, CACHE.ADR, NUM, SIMP.CODE). This procedure performs a non-incrementing, simplex high read. It combines the functions of SREAD and NREAD.

SNREAD is typically used to access FIFO buffer in the I/O port.

(11) READL (SYS.ADR, CACHE.ADR, NUM). This procedure performs a long read operation across page boundaries. Since it must check the arguments in order to break up the transaction, there is considerable software overhead associated with reading system memory this way.

(12) WRITEL (SYS.ADR, CACHE.ADR, NUM). This procedure writes into system memory across page boundaries.

(13) HOG.BUS. This procedure increments the cache global variable HOG.WORD. It does not perform a bus transaction. However, next time a bus transaction is performed, the bus will be "hogged" (not released) at the end of that transaction if HOG.WORD is non-zero.

(14) RELEASE.BUS. This procedure decrements HOG.WORD and releases the bus, if the HOG.WORD is zero, by doing a dummy bus transaction. If HOG.WORD is zero on entry to this procedure (an illegal condition), an illegal op code is executed, thereby trapping the triad in illegal op code interrupt handler. This feature has been provided solely as a means of debugging software.

(15) ARELEASE.BUS. This procedure unconditionally releases the bus. The HOG.WORD is initialized to zero and if holding the bus it is released by doing a dummy bus transaction.

(16) SYNC. This procedure is part of the processor synchronization process. It acquires the bus and then releases it if the bus was not being held on entry to this procedure. Otherwise, it releases the bus and re-acquires it. In either case, the result is that all participating member processors are forced to do a bus poll.

Procedures 1 to 10 use the same core with different entry points. The different entry points set up the high order 5 bits of the bus address and certain fields of the command word in two local variables. The core routine uses these two words along with the three procedure arguments to load various SBA (System Bus Access) registers. HOG.WORD is

checked to determine if the bus should be held or released after the transaction and the SBA.CMND (command register) loaded appropriately to begin the bus transaction. Writing into the command register initiates the bus transaction upon completion of which control returns to the next instruction. All interrupts are masked at the start of the routine and the interrupt mask is restored to its original status at the end.

Procedures 11 and 12, READL and WRITEL, use a slightly different core routine. Here, the required bus transaction is broken down into smaller parts, none of which crosses a cache or system page boundary. Interrupts are masked during these procedures also.

Finally the last four procedures use a single core routine that performs a simple dummy bus transaction. In the dummy transaction one word is transferred from system memory address zero to cache location "4000" (which does not exist). Interrupts are masked during this transaction.

3.7.2 Error Latch Service Routines

Although error latches can be read using service routines described in the previous section, it is possible to write even more efficient routines that are customized for error latches.

The first of these is READ.EL. This AED procedure is called every R1 frame by SCC (System Configuration Control) program. The function of this procedure is to read all 48 error latches of 12 LRU's. The READ.EL routine obtains R bus assignments of each LRU from system memory, sets up simplex read select bits and then calls an assembly language procedure ERRLATCH to actually do the bus transaction.

ERRLATCH is called with two arguments: (1) N, the LRU from which to read the latches, and (2) select bits for the simplex read. All four latches of LRU N are read and the four values are written in a predetermined place, at MM.ERR.LATCH(4*N), in the system memory.

The last procedure is CLR.ALL.ELS. This is an AED procedure that clears all error latches by reading them. It uses SREAD service routine for this purpose. This procedure is called only infrequently by the configuration controller.

3.7.3 Timer Routines

The timer routines keep track of several interval timers in software. There is only one hardware internal timer in each processor. This timer is continually decremented every 250 microseconds. When the timer decrements from zero to -1, a timer interrupt is generated. The interval timer, with the help of timer interrupt, is used to initiate a new R4 frame and to place maximum execution time limits on applications tasks. Since a number of such tasks can be in progress simultaneously within a triad, several interval timers may be active simultaneously. The following routines are used to track various interval timers in software and to take appropriate action when a timer interrupt is generated. Their source code is contained in FTMP.AED(TIMERS) and the compiled output appears in FTMP.LIST(TIMERS).

TIMER.INIT. This procedure clears timer interrupt, sets interval timer to its maximum value (16 seconds) and initializes the global variables used by other timer routines. It is called from the R4 dispatcher once after a triad starts up.

START.R4.TIMER(I). This procedure is called by the R4 dispatcher just before activating an R4 applications task. It disables all interrupts, sets the interval timer to I, clears timer interrupt and sets timer flags R4.TIMER.ARMED and R4.TIMER.ON to TRUE.

STOP.R4.TIMER. This procedure is called by the R4 dispatcher just after completion of an R4 applications task. It sets the interval timer to 16 seconds, clears timer interrupt, sets timer flags R4.TIMER.ARMED and R4.TIMER.ON to FALSE and restores the interrupt mask to that saved by START.R4.TIMER.

START.R3.TIMER (I). This procedure is called by the R3 dispatcher just before starting an R3 applications task. It saves the current interrupt mask, disables all interrupts and sets R3.TIMER.ON to TRUE.

If the interval timer is not already loaded with a useful value (R4 frame interrupt time) then it is loaded with I and appropriate timer flags are set after clearing the timer interrupt. If, however, the timer is already armed for the next R4 frame then the shorter of I and time to next R4 frame is loaded and appropriate flags set in cache.

STOP.R3.TIMER. This procedure is called by the R3 dispatcher just after completing an R3 applications task. It restores the interval timer to its status when START.R3.TIMER routine was called. For example, if the timer was not armed with anything at that time then it is set to 16 seconds. If it was armed for R4 frame time then the time remaining to the start of the next R4 frame is loaded in the timer. The mask is restored to R3 dispatcher status.

START.R1.TIMER (I). This procedure is called by the R1 dispatcher just before starting an R1 applications task. Its functions are identical to that of START.R3.TIMER.

STOP.R1.TIMER. This procedure is called by the R1 dispatcher just after completing an R1 applications task. Its functions are identical to that of STOP.R3.TIMER.

HOLD.R3.R1.TIMERS. This procedure is called by the R4 dispatcher upon entry. If the timer is loaded with either the R3 or the R1 applications task time limit then the time left for the task is saved. The interval timer, in any case, is set to 16 seconds and timer interrupt cleared.

HOLD.R1.TIMER. This procedure is called by the R3 dispatcher upon entry. If the R1 applications task timer is running, it is halted and time left for the R1 task is saved. If this triad is R4 responsible and the interval timer is not already armed with the R4 frame time then it is loaded with the R4 frame time.

RELEASE.R3.R1.TIMERS. This procedure is called by the R4 dispatcher just before exit. The R3 or R1 task timers that were frozen upon entry by the R4 dispatcher are released by this routine. In addition, if the R4 frame timer is running, the interval timer is loaded with the time left to the next R4 frame.

RELEASE.R1.TIMER. This procedure is called by the R3 dispatcher just before exit. It releases the R1 task timer if it was running upon entry.

TIMER.INT.HANDLER. This procedure is entered in response to a timer interrupt. It pends the R4 process in the PSD chain if it is an R4 frame interrupt. The type of interrupt is ascertained by the timer flags which are set at the time the interval timer is armed. If the interrupt is caused by an application task exceeding its allotted time then that task is purged from the PSD chain by calling PURGE.APSD procedure.

PURGE.APSD. This procedure is called by the TIMER.INT.HANDLER. It removes from the PSD chain the process pointed to by the RESUME.PNTR of the active process. This is done simply by changing the RESUME.PNTR of the active PSD.

3.7.4 Miscellaneous Primitives

There are five functions collected together under this category. These are described in the following.

(1) LOCK(X). This procedure locks a word, X, in system memory. It does this by storing the triad id in the three least significant bits of X, provided X was unlocked, that is, it was zero. If X is already locked by another triad, then this procedure stores the triad id in the next 3 bits and the EXEC.LEVEL of the requesting triad in the next two bits. If another triad is already waiting for X to become available, this triad puts in a request in bits 8 to 12. (Figure 3-54 shows the structure of the lock word.) Having put in a lock request, it waits for an unlock message in IPC register 0. When the bit in IPC R0 corresponding to EXEC.LEVEL is set, then x is available. (See discussion below in UNLOCK). Figure 3-55 shows the N-S diagram of the lock procedure.

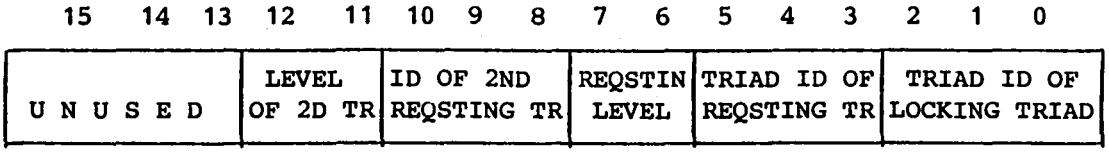


Figure 3-54. Lock word structure.

(2) UNLOCK (X). This procedure unlocks word X in system memory by clearing its triad id from the low order 3 bits of X. If another triad is waiting for X, it sends its IPC register 0 unlock message and also shifts X right 3 bits thereby locking X for the waiting triad. The unlock message is really a bit being set in IPC R0 corresponding to the EXEC.LEVEL. That is, if the lock request was made at level 2 the bit 2 in IPC R0 will be set to 1. The lock procedure waits for the unlock message by continually checking this bit to see if it is set. Since a triad can request a resource at R1 or R3 level, be interrupted and request another resource at another level, it is necessary to make sure that unlock messages at lower levels are not lost at higher levels. Therefore a copy of IPC R0 for each task is kept in system memory. When it is necessary to send an unlock message to a triad, a copy of its IPC R0 is read from system memory, appropriate bit set in it, the updated word written back to system memory and finally the same word is sent to the triad. This, in a way, OR's the present unlock message with any previously pending UNLOCK messages. The lock routine similarly clears only one bit from the system memory copy ofits IPC R0, the bit that corresponds to its requesting level. Figures 3-56 and 3-57 show the structure of the lock word X before and after being unlocked. Figure 3-58 shows N-S diagram of UNLOCK.

(3) SET.BIT (VALUE, BIT.NO, WORD). This procedure sets bit BIT.NO in the system memory address WORD to VALUE in one unbroken bus transaction. The word is read from system memory with the bus being hogged, the BIT.NO bit is set to VALUE, the word written back to system

<table>
<tr><td colspan="4">DISABLE INTERRUPTS WHILE BUS IS BEING HOGGED.
HOG BUS.
TEMP VARIABLE LOCKED = FALSE. (X IS NOT YET LOCKED BY THIS TRIAD.)
READ X (WORD TO BE LOCKED) FROM SYSTEM MEMORY. STORE IT IN TEMP.X.</td></tr>
<tr><td colspan="4">IS X UNLOCKED (= 0)?</td></tr>
<tr><td>YES</td><td colspan="3">NO</td></tr>
<tr><td rowspan="3">X IS AVAILABLE.
LOCK X BY STORING TRIAD ID IN LOW ORDER 3 BITS OF X
LOCKED = TRUE.
(X IS NOW LOCKED)</td><td colspan="3">IS X ALREADY LOCKED BY THIS TRIAD?
(LOW ORDER 3 BITS OF X = TRIAD.ID)</td></tr>
<tr><td>YES</td><td colspan="2">NO</td></tr>
<tr><td>LOCKED= TRUE</td><td colspan="2">IS ANOTHER TRIAD WAITING FOR X?</td></tr>
<tr><td></td><td>YES</td><td>NO</td></tr>
<tr><td></td><td>STORE TRIAD ID & LEVEL 'L' IN BITS 8-12 OF TEMP.X.</td><td>STORE TRIAD ID & 'L' IN BITS 3-7 OF TEMP.X.</td></tr>
<tr><td colspan="4">WRITE TEMP.X IN X.
RELEASE BUS.
ENABLE INTERRUPTS.</td></tr>
<tr><td colspan="4">IS X LOCKED BY THIS TRIAD? (LOCKED = TRUE?)</td></tr>
<tr><td>YES</td><td colspan="3">NO</td></tr>
<tr><td>NIL</td><td colspan="3">WHILE NOT LOCKED (I.E. DO UNTIL LOCKED)</td></tr>
<tr><td></td><td colspan="3">IS THERE AN UNLOCK MESSAGE? (BIT IN IPC REGO SET?)</td></tr>
<tr><td></td><td colspan="2">YES</td><td>NO</td></tr>
<tr><td></td><td colspan="2">DISABLE ALL INTERRUPTS.
HOG BUS.
READ SYSTEM MEMORY COPY OF IPC REG0.
'AND' BIT CORRESPONDING TO LEVEL 'L' TO ZERO.
WRITE IPC REG0 OF THE SELF TRIAD.
RELEASE BUS.
LOCKED = TRUE.
ENABLE INTERRUPTS.
X NOW LOCKED BY THIS TRIAD.</td><td>NIL</td></tr>
</table>

Figure 3-55. N-S diagram of lock.

15 14 13	12 11	10 9 8	7 6	5 4 3	2 1 0
U N U S E D	LEVEL OF 2D TR	ID OF 2ND REQSTING TR	REQSTIN LEVEL	TRIAD ID OF REQSTING TR	TRIAD ID OF LOCKING TRIAD

Figure 3-56. Lock word before being unlocked.

15 14 13	12 11 10 9 8	7 6	5 4 3	2 1 0
U N U S E D	0 0 0 0 0	LEVEL OF 2 TR	TRIAD ID OF 2ND TRIAD	TRIAD ID OF 1ST REQST TR

Figure 3-57. Lock word after being unlocked.

DISABLE INTERRUPTS WHILE BUS IS BEING HOGGED. HOG BUS. READ X (WORD TO BE UNLOCKED) FROM SYSTEM MEMORY. STORE IT IN TEMP.X. SHIFT RIGHT TEMP.X BY 3 BITS.	
IS ANOTHER TRIAD WAITING FOR X? (TEMP.X ¬= 0?)	
YES	NO
OBTAIN LOCK REQUEST LEVEL FROM X. SEND WAITING TRIAD'S IPC R0 UNLOCK MESSAGE. (SET ONE BIT CORRESPONDING TO LEVEL IN R0). GET RID OF 2 LEVEL BITS FROM TEMP.X.	NIL
WRITE TEMP.X IN X. RELEASE BUS. ENABLE INTERRUPTS.	

Figure 3-58. N-S diagram of unlock.

memory and the bus is then released. Interrupts are masked in this procedure.

(4) IPC.INT.HANDLER. This procedure handles IPC interrupts. Writing to IPC registers 2 and 3 causes an interrupt. Presently only R2 is used by the Executive. IPC interrupts are used for three purposes.

(a) PEND R4 PSD: This is signalled by setting bit 0 of IPC R2.

(b) PEND RESTART PSD: This is signalled by setting bit 1 of IPC R2.

(c) HALT: This is signalled by setting bit 2 of IPC R2. The last one is provided only for debugging purposes.

Since one or more interrupts could be pending when the IPC R2 is written to cause another IPC interrupt, a copy of IPC.R2 is kept in system memory. The IPC.INT.HANDLER clears this to zero and responds to all the pending interrupts as indicated in IPC.R2.

(5) KICK (TRIAD.TRACKER). This procedure examines the kick field (bits 8 - 11) of the argument (TRIAD.TRACKER). It sends an "Pend R4" IPC interrupt to a triad whose bit is set in the kick field. It then resets that bit indicating that triad has been kicked. To send the IPC message, acopy of IPC.R2 of the target triad is read from system memory. The "Pend R4" bit is set in this word, the updated word written back into system memory and also into IPC R2 of the target triad. The bus is "hogged" during this operation.

CHAPTER 4

FACILITIES SOFTWARE

On the ground, the FTMP is supported by a test adapter and a PDP-11. This chapter describes the ground support software resident in the PDP-11.

4.1 CTA

The test adapter is a front panel-like keyboard interface to the the FTMP. It can be connected to an LRU through the processor region transfer bus. At the same time, the test adapter can also be connected to the PDP-11 on the Unibus. A program on the PDP-11, called CTA (Collins Test Adapter program), provides on any PDP-11 terminal all the functions that are available from the test adapter keyboard. These functions or commands relate to the processor and the RAM of the master LRU, that is, the LRU to which the test adapter is connected. Specifically, these commands are Halt, Run, Reset, List, and Set. The functions performed by these commands are self-explanatory.

Since the test adapter does not have access to the FTMP system bus, it can not access any device that is on the system bus but not on the processor transfer bus. Therefore it can not directly access the

system memory, system control registers, I/O port registers, etc. CTA, however, can be used to access the system bus devices indirectly as described in the following.

A cooperator program, COOP, is loaded from the PDP-11 into the master LRU cache. 'Load' is one of the functions performed by CTA. CTA sends device access request to COOP which performs the operation and returns the results if any to CTA. The exact protocol is as follows.

Four words in cache RAM (2001-2005) are reserved for CTA-COOP communication as shown below.

2001 Command

2002 Address High

2003 Address Low

2004 Data

There are 3 commands. Read, Write, and Resume. The first two are used to read or write a system bus device. The last command, resume, instructs the COOP process to resume the next process in the PSD chain. Only one word is transferred to/from a system bus device in one transaction between CTA and COOP. This communication, however, is transparent to the user. Mnemonic commands are provided to access all the relevant registers etc. in the FTMP. The FTMP system bus addresses of these registers have been programmed in CTA. The commands and their functions are as follows:

MLIST page offset num: This command is used to list on the terminal 'num' number of words from system memory starting at address as defined by the page and the offset. This multiword transaction is broken up into 1 word transactions by CTA.

MSET page offset = data: This is used to set a system memory location whose address is defined by page and offset to 'data'.

CCR n, m = abcd: This sets control register m of LRU n to abcd.

PE n = (x,y,z): This enables LRU n on P buses x, y, and z by writing into the two BGU registers of LRU n. There are similar commands, (RE, TE, and CE) to enable an LRU on R, T, and C buses.

PS n = (x,y,z): This writes into the P select register of LRU n to listen to buses x, y, and z. There are similar commands (RS, TS and CS) to select R, T, and C bus sets.

MRR n = abcd: This sets memory relocation register (MRR) of LRU n to abcd.

HOG: The master LRU 'hogs' the system bus next time it does a bus transaction.

UNHOG: The master LRU releases the system bus.

RTC: This command is used to read the FTMP Real Time Clock.

LOAD file: This command is used to load an absolute load module from the PDP-11 into the FTMP cache and the system memory.

4.2 PROM Programmer

A program called PROM is available on the PDP-11 to program the FTMP PROM's. Each LRU in the FTMP has an 8k PROM card which contains 8 "2716" PROM's. The hardware to program all PROM's on such a card is attached to the PDP-11 Unibus. The PROM program listens to the following commands.

LOAD file: This is used to load the PROM section of an FTMP absolute load module into the internal buffer of the program. A letter at

the beginning of each record in the load module indicates whether that record is destined for the PROM, RAM, or the system memory.

EDIT: This invokes a simple editor which can be used to examine or alter the load file in the buffer.

READ: This transfers the contents of the PROM into the internal buffer.

INIT: This sets the internal buffer to all '1's which is the state of an unprogrammed PROM.

PROGRAM: This programs the PROM with the contents of the internal buffer.

COMPARE: This compares the PROM contents with the buffer contents and lists any differences found.

Typical programming time is about 40 milliseconds per word or about 6 minutes for the full 8k board.

4.3 Fault Injector Software (FIS)

The fault injector software (FIS) package resident on the PDP-11 provides commands at a PDP-11 terminal to perform all the functions necessary to inject faults into LRU 3 of the FTMP and observe the results.

The fault injection hardware is attached to the PDP-11 Unibus on one end. It consists of six cards, A, B, C, D, E, and F, each of which can be interfaced to 8 pins of an IC package on the FTMP. Signals, simulating faults, can be injected on any combination of these 48 pins as directed by FIS commands. The results, that is, fault detection, identification, recovery times etc., are transmitted from the FTMP to FIS on a

1553 bus. This I/O function in the FTMP is performed at R4 frequency in the R4 dispatcher prolog.

The FIS commands and their functions are as follows:

DEFINE Unn M: This command defines an M pin IC package whose location on the circuit board is Unn.

MAP n Am ℓ: This maps pin n of the device defined in the previous command into pin m of board A of the fault injector. ℓ-1 subsequent device pins are mapped to ℓ-1 subsequent A board pins.

DESCRIBE n abcd: This defines the fault (abcd) to be injected on pin n of the device. Part of the fault description tells FIS whether the fault should be injected as input to the device pin or as output of the device pin and into the corresponding socket pin.

ENABLE n: This selects pin n of device for fault injection, when a fault is actually injected next time.

DISABLE n: This deletes pin n of device for fault injection, when a fault is actually injected next time.

EXEC: This injects the fault(s) as defined by the previous commands. The fault is asserted either until FTMP has recovered from the fault or 10 seconds whichever happens first.

AUTO n: This command repeats the EXEC function n times. However, before injecting a fault a 'GET READY' command is sent by FIS to the FTMP. The FTMP in response to this command repairs LRU 3 subunits and brings them on-line in active state. A 'READY' signal is sent back by the FTMP to FIS at this point so that the fault may now be injected in LRU 3. (See Section 3.5.2 for a detailed discussion of the FIS-FTMP

communication protocol). After each event, the results received from the FTMP are recorded. At the end of the run the collected results can be displayed in various histogram formats.

4.4 CAPS Simulator

This simulator which runs on the PDP-11 in an interactive mode was used to debug the FTMP software prior to the hardware delivery.

The CAPS simulator basically provides a uniprocessor environment. Some of the multiprocessor aspects of the FTMP are also simulated in this package. Specifically, a single CAPS-6 processor, 12k cache memory and 48k system memory is simulated. The simulated processor includes the CAPS-6 instruction set, interval timer and power on clear, illegal op-code, timer and IPC interrupts. In addition to one processor and cache, system and control registers of other 11 LRU's are also simulated. These include the BGU's, Error Latches, SCU registers and the Real Time Clock. The System Bus Access (SBA) control registers are available only in the main LRU, that is, the one with the processor and the cache.

Breakpoint, single instruction step and normal execution modes are provided to facilitate software debugging. The CAPS control register contents, unlike in the FTMP, can also be displayed. These include TOS, LENV, SPCR, etc. The Real Time Clock and the interval timer can also be frozen at a breakpoint unlike in the FTMP. The state of the simulator which includes the processor, cache and system memory contents can be saved in the PDP-11 during any simulator run. Several such states can be saved subject to storage availability on the PDP-11 disk. The simulator

can be initialized either with a new load file or with a previously saved snapshot of the simulator.

The simulator operates at approximately one fifth the speed of the CAPS processor.

4.5 Miscellaneous Facilities Software Packages

There is a TSO link that connects PDP-11 to the Amdahl 470 Time-share Option (TSO) network. A TSO program in the PDP-11 supports the link. Using this software package any PDP-11 terminal can be transferred into a TSO terminal. In addition, any files from the Amdahl can be downloaded into the PDP-11 using the TSO link. This facility is used to transfer FTMP load modules from the Amdahl where they are created to the PDP-11 from where they are loaded into the FTMP.

There is a 1553 emulator card support software called M15. M15 is similar to CTA. It provides commands at a PDP-11 terminal to control 1553 emulators. Programs can be loaded and run in the emulators using commands similar to the CTA commands.

The 1553 emulator software consists of two microcode programs. One supports FTMP-1553-RS232 communication while the other supports FTMP-1553-PDP11 communication. Their functions are as follows.

The RS232 interface is used to display FTMP status and other FTMP information on an HP terminal and transmit operator commands typed on the FTMP console back to the FTMP. The emulator microcode that supports these functions receives input from the console through a UART. It sends

the console input to the FTMP when requested to do as by an I/O driver (TTY) in the FTMP. It receives display data destined for the console from the FTMP and transmits it to the console at 4800 baud. Internal buffers are maintained in the emulator for both the input and the output. The details of the communication protocol between the microcode and the FTMP have already been described in Section 3.6.

The other 1553 emulator microcode package is used to emulate a 1553 remote terminal with 32 sub-addresses. This code communicates with an I/O driver (SENSOR.IO) in the FTMP. It transmits aircraft sensors, navigation and guidance data, FIS commands etc. to the FTMP when requested by SENSOR.IO. This information is obtained by the microcode from the PDP-11 memory through DMA. It also receives actuator commands and data destined for FIS from the FTMP and places them in the PDP-11 buffer areas through DMA.

Finally, there is a PCL-11 driver in the PDP-11 to communicate with the 707 simulation running in the dual VAX-11 computers. The PCL-11 is a high speed parallel data link between the VAX1, VAX2, and the PDP-11. This link is used to transmit sensor, actuator, navigation, and guidance information between the 707 simulation and the PDP-11 buffers used by the emulator microcode.

CHAPTER 5

ACCEPTANCE TEST/DIAGNOSTIC SOFTWARE

A set of acceptance test programs was written at Collins Avionics and at CSDL to verify the FTMP hardware operation prior to the delivery of the hardware to CSDL. These programs have since been modified and expanded to extend their scope to more areas of the hardware. Basically, each of these diagnostic programs runs in a stand-alone mode. The Kernel program which is resident in the PROM of each LRU has provisions to start up a program in the processor cache memory or to load the cache from the system memory and then start up that program. (See Section 3.2 for a discussion of Kernel).

Typically, the LRU under test is enabled on all the P, T, and C buses by inserting the shorting plug into it. Appropriate test program is loaded into its cache RAM or the system memory if several LRU's are to participate in the test. In the latter case, each LRU under test in turn is enabled on all P and T buses and its CPU control register R3 is set to 2. Each processor that is so started copies system memory locations (2005-2007 and 2010-3FDF) into corresponding cache memory locations and goes into a wait loop until register R3 is cleared. Locations 2005-2007 contain the TOS, STKLM, and SPCR of the diagnostic program. Each

processor then starts executing this program when its control register R3 is cleared to zero.

Individual command lists have been written to set up the system correctly for each diagnostic test. These command lists are executed from CTA (see Section 4.1) in an interactive mode. The command lists prompt the operator to set up any cache memory locations, control registers, etc. that are dependent on the slot(s) under test and to remove the master plug, if necessary. The results of the diagnostic tests are shown on the screen and/or on the test adapter display windows which are also setup by the operator to display the addresses of interest for each test.

Following is a brief description of each of the diagnostic programs.

5.1 LRU Diagnostics

This is one of the most comprehensive FTMP diagnostic programs. It runs in one LRU and checks just about all LRU components except the cache and the processor. A PDP-11 command file LRUDIAG.CMD can be used to load and start this program in any LRU. The command file prompts the operator to input the slot ID of the LRU under test.

The LRU diagnostic program calls a number of procedures to test various LRU components. First of all, INIT is called to initialize various parameters including the LRU ID. Next, XMIT5 is called to configure the LRU to transmit on all 5 bus sets. The Real-Time Clock (RTC) is the first item to be checked. Two procedures RTC1 and RTC2 are called to write and read the RTC while the operator checks the waveform. Then

RTC3 is called which reads the RTC and displays it in the test adapter windows. RTC3 checks the clock frequency against the interval timer. It verifies that each RTC bit toggles. It also verifies certain read/write features of the RTC. For example, RTC should not count until its high word is written into after a low word write. Also when the low word is read the high word at the time should be latched.

The next procedure to be called is BOX ID. This program verifies that the LRU responds to its LRU ID and only to its LRU ID. To do this CPU control register Rϕ in all the LRU's is written over the system bus and the self Rϕ is read over the transfer bus to verify that Rϕ is as expected. If all the tests so far are successful, the operator is prompted to remove the shorting plug.

DISABLE1 is then called. This procedure disables the LRU on one bus line at a time and verifies that the proper bus line is disabled. The next procedure is XMIT3 which configures the LRU to transmit and receive on the default 3 buses. Following this configuration ERRLATCH is called. This procedure clears all error latches by reading them once. It then reads them again to verify that they did clear.

The next procedure to be called is BUSERR. It simulates errors on each of 20 bus lines by transmitting on appropriate bus sets. The expected errors are verified by reading the error latches. The next procedure, RTC4, tests the arm/disarm bit of the RTC. The Real-Time Clock should respond to write requests all the time but to read requests only if it is armed. The next procedure, MRR1, sets the memory relocation register (MRR) to zero and verifies system memory read/write operations.

The next test checks the system memory operation. A procedure MEMDIAG writes 0, FFFF and address to each location in the 16k system memory contained in the LRU under test. Read/write operations are also conducted across page boundaries to see that they are terminated correctly. The next procedure, MRR2, sets the MRR to all possible relocation values and verifies that the memory responds only to the relocated address space.

The next LRU component to be tested is the IPC (Inter-Processor Communication) register set. A procedure, IPC, verifies the IPC register operation by writing to them on the system bus and reading them back on the processor transfer bus. Registers 2 and 3 are also checked to verify that writing to them generates IPC interrupts.

The next procedure, IO, checks the operation of the I/O port in the LRU. The 32-word FIFO in the I/O port is loaded with data and the command register is loaded with a 'transmit 32 word' command. Status word is checked to verify that there were no transmit errors. Receive function of the port is similarly verified.

The last test is performed by SBC. This procedure checks the operation of the system bus controller. Various error conditions such as page boundary crossing etc. are simulated and status register checked to verify that the condition was detected. Bus hog operation, effect of static priority or bus poll, etc. are also checked.

If the LRU passes all the tests a word "A1 LRU ID" is displayed in one of the test adapter windows. If it fails any test, the program is terminated and "2BAD" is displayed.

5.2 Opcode Diagnostics

A procedure, OPPROG, tests all instructions in the CAPS-6 processor except those related to the interrupt system. Each instruction is executed with opcode on even and odd byte boundary and for all hardware stack conditions. If the processor passes all tests "A1A1" is displayed. If not, "2BAD" is displayed. The procedure OPPROG is based on a table lookup strategy. A data set, OPDATA, contains initial and final stack conditions and expected results for each instruction execution. This table occupies more than 4k bytes.

5.3 Interrupt Diagnostics

Interrupts and interrupt related instructions are checked by a program called INTPROG. This program verifies POC, INTRTN, HALT, ASNMSK, REFMSK, SWPMSK, Interval Timer interrupt, PM, and PI fault interrupts. The mapper is also checked to verify correct mapping operation as well as correct read/write of the mapper itself. INTPROG uses tables setup in the data set INTDATA. A successful test is terminated by "A1A1" being displayed on the test adapter. "2BAD" is displayed if any of the tests was a failure.

5.4 Cache Memory Diagnostics

The cache RAM is tested by a procedure called CMEMDIAG. There are two versions of this program, low and high. The low version is loaded in the high part of cache and tests the lower half of the memory. The high version tests the higher half. Several tests are performed. Each

location is written with its address and read back to verify the operation. Following this test various patterns, such as 100100100 ..., are written throughout the memory. The patterns are varied to 01001001..., 001001001..., etc. and their complement. These tests should discover any stuck-at-0, stuck-at-1 and 'slowly dying' bits, that is, bits that change when surrounded by all 1's or all 0's. Stuck address and data lines should also be uncovered by these tests.

5.5 Synchronization and Multiprocessing Diagnostics

The diagnostic programs described so far have been designed to run in a single LRU. A procedure, SYNC, checks the ability of processors to synchronize themselves to each other and for several triads to compete for the system bus.

The SYNC program clears all interrupts, resets the interval timer and initializes SBA Control register. It then goes through a synchronization procedure as described in Section 3.3. Once three processor members of the triad have been synchronized, the error latches are cleared by reading them. Following this, all 48 error latches from 12 LRU's are read once again and written to the system memory after being reduced to 4 words. 12 error latches of each type (P, R, T, and C) are OR'd together to form an error word for that type of bus. The interval timer is then armed to go off in 8 seconds. During that time the Real-Time Clock is read and written into the system memory. Eight seconds later, a timer interrupt is generated as a result of which Kernel transfers control to the timer interrupt handling routine.

The timer interrupt handler reads 48 error latches, converts them to 4 error words as described before and writes them to another system memory location. PSD (Process State Descripter) of the interrupted process, that is, the main program is also saved in the system memory. (See Section 3.2 for a description of Kernel and PSD). The interrupt mask is then changed to allow IPC interrupts and an IPC interrupt is caused by writing IPC register R2 over the system bus. The Kernel transfers control to the IPC interrupt handling routine.

The IPC interrupt handler is similar to the timer interrupt handler. It reads, condenses, and writes the error latches to system memory and it also saves the timer PSD in the system memory. Four seconds later, as timed by the interval timer, it 'Resumes' thereby returning control to the timer interrupt handler. That process, in turn, waits 4 seconds and Resumes the main program after setting the interval timer to go off in 8 seconds. This process repeats every 24 seconds.

The error words, as recorded by the three processes, can be examined in the system memory to reveal any bus errors. If the processors can synchronize after a power on reset and stay synchronized through timer and IPC interrupts, the error words should stay clear. If they are not, various PSD's can be examined to determine the exact value of the instruction counter (SPCR) at which point each processor received the timer/IPC interrupt.

There are three command files, SYNC1.CMD, SYNC2.CMD and SYNC3.CMD. They start up one, two, and three processor triads with the SYNC program, respectively. The SYNC program writes error words and

PSD's in a system memory location indexed by triad ID. That is, in a multiprocessor situation, each triad saves its results in a different system memory location.

The SYNC program thus verifies processor synchronization, bus arbitration logic and timer and IPC interrupts in a multiprocessor mode.

5.6 Clock Diagnostics

A procedure, CINMUX, tests all paths through C bus inmuxes of all LRU's present and all possible clock quads.

First of all, the program determines which LRU it is running on by writing to the CPU control register Rϕ of all the LRU's on the system bus and then reading its own register on the transfer bus. It then enables itself on all the buses and prompts the operator to remove the shorting plug. Once that is done, it determines which slots are actually populated with LRU's by reading their C error latches.

From amongst the LRU's present, four are chosen to form an initial clock quad. This clock quad is enabled to transmit on all possible three and four C bus combinations. In the former case, one of the four clock elements is not enabled on any C bus. The error latches are cleared after each clock reconfiguration and after a wait of 50 milliseconds they are reread and compared to expected values. If no disagreement is found the clocks are reconfigured to the next bus combination. Forty bus combinations are tested for each clock quad.

This process is repeated for all possible clock quads that can be formed from the LRU's present. The test continues as long as no errors

or unexpected errors are found. The program halts and displays the four clock elements and their bus assignments if an unexpected error is discovered.

This Page Intentionally Left Blank

CHAPTER 6

APPLICATIONS SOFTWARE

The FTMP applications software consists of three major segments, console software, flight control, and navigation. These are described in the following three sections.

6.1 Console Software

The FTMP console is used to monitor the FTMP status and to input operator commands to the FTMP. The console is an HP terminal which is driven by the FTMP through a 1553 bus and a 1553-RS232 interface. (See Sections 3.6 and 4.5 for the details of the interface).

There are four different displays. These are as follows.

STATUS This display is the default system display. It can be invoked by the console command STATUS. This display shows the status of each processor, system memory, and bus line in the FTMP. Each processor and system memory module as identified by the slot number (0 to B) may be in one of three states, active, spare, or failed. For active units, the triad number is also shown. Each bus identified by the line number (1 to 5) and type (P, R, T, or C) is also shown to be active (A), spare (S) or failed (F). In addition, the four members of the clock quad are shown by

their slot numbers. The time of day is shown in all the displays. Operator commands are echoed on the bottom line of the screen. The status display is shown in Figure 6.1.

BUS ASSIGNMENTS This display shows the buses on which each LRU is enabled to transmit and the bus set selected by each LRU to receive. This display is invoked by the command BUS. The bus display is shown in Figure 6.2.

FAILURE LOG This display shows the time of day (in hours, minutes and seconds) of failure for all the failed units in the system. A failure reason code accompanies the failure time. This display is invoked by the command LOG. The log display is shown in Figure 6.3.

TRANSIENT ERROR LOG This display shows the transient fault-index for each unit in the FTMP. It is invoked by the command TRLOG. The transient log display is shown in Figure 6.4.

INPUT/OUTPUT This display shows the status of the 1553 I/O ports in the FTMP. Through these ports, the FTMP interfaces with the display console, the fault injector, the PDP-11/60, and the aircraft simulation facility. The I/O display shows the ports through which these devices are being accessed at any given time. It is invoked by the command IO. A diagram of the I/O display is shown in Figure 6.5.

AUTOPILOT STATUS This display shows the status of the simulated aircraft in terms of altitude, airspeed, heading, attitude, and attitude rate (pitch, roll, and yaw rates). It also shows the autopilot/flight director mode of operation and the pilot inputs to the autopilot and the

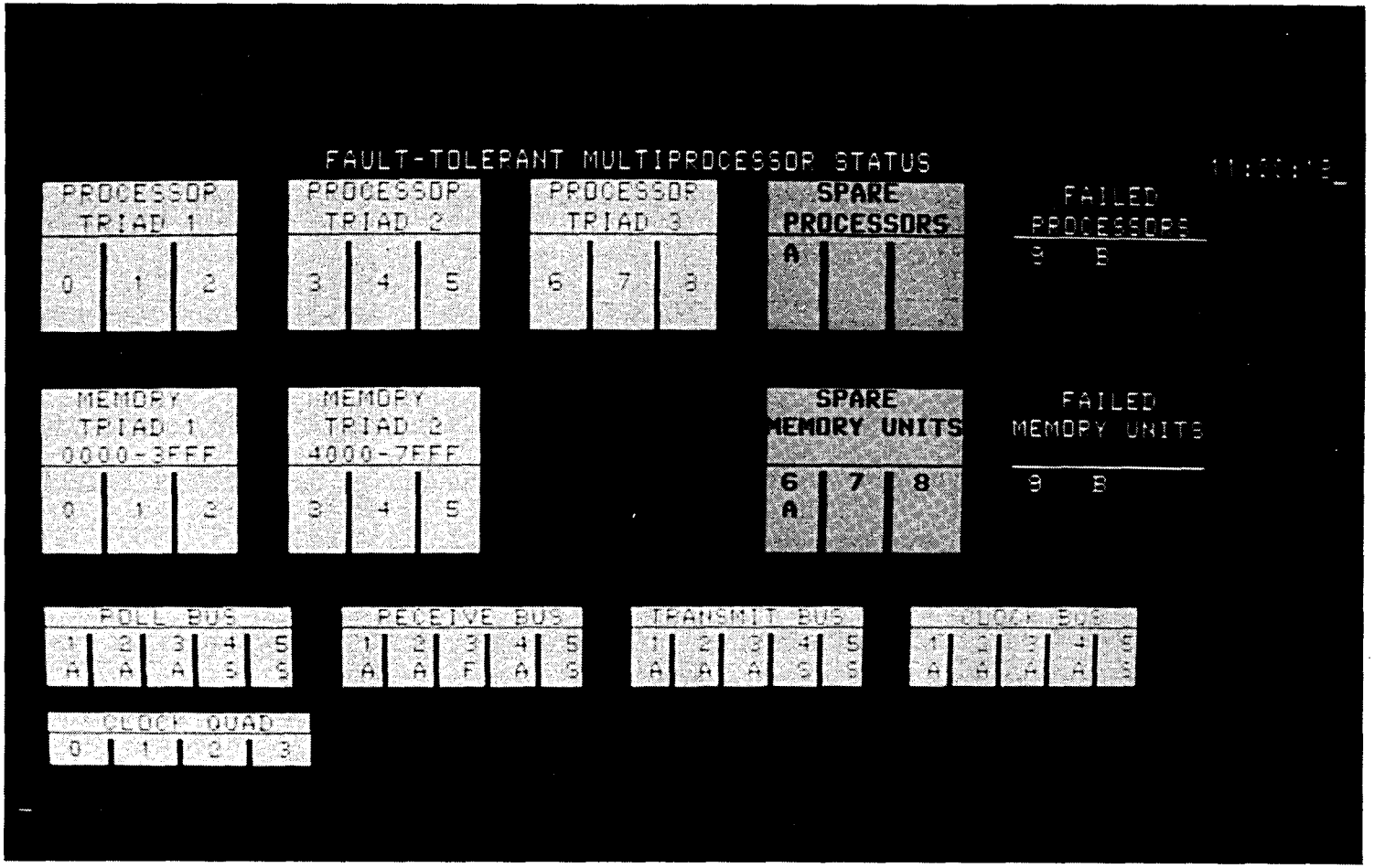


Figure 6-1. System Status Display.

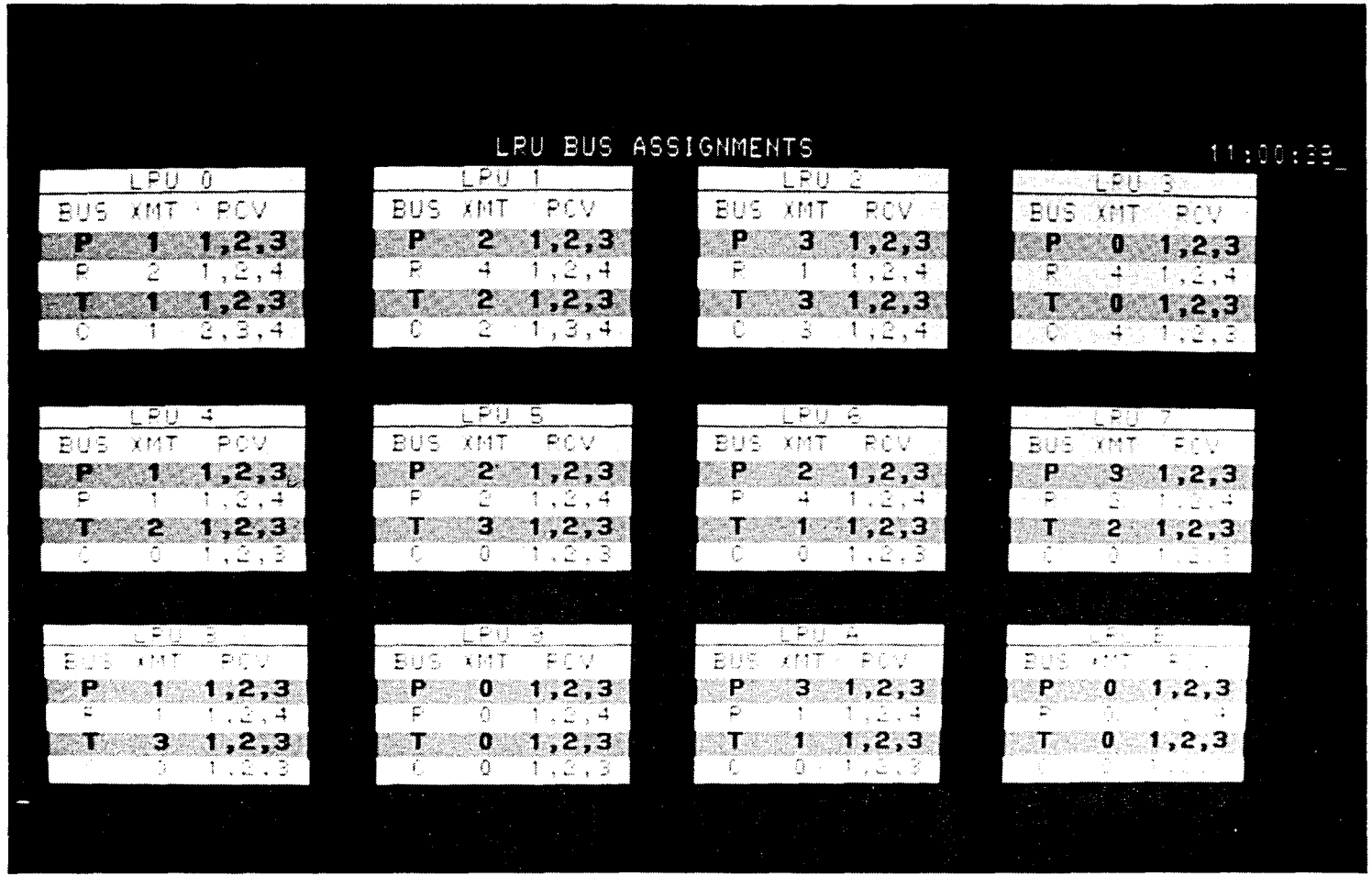


Figure 6-2. LRU Bus Assignment Display.

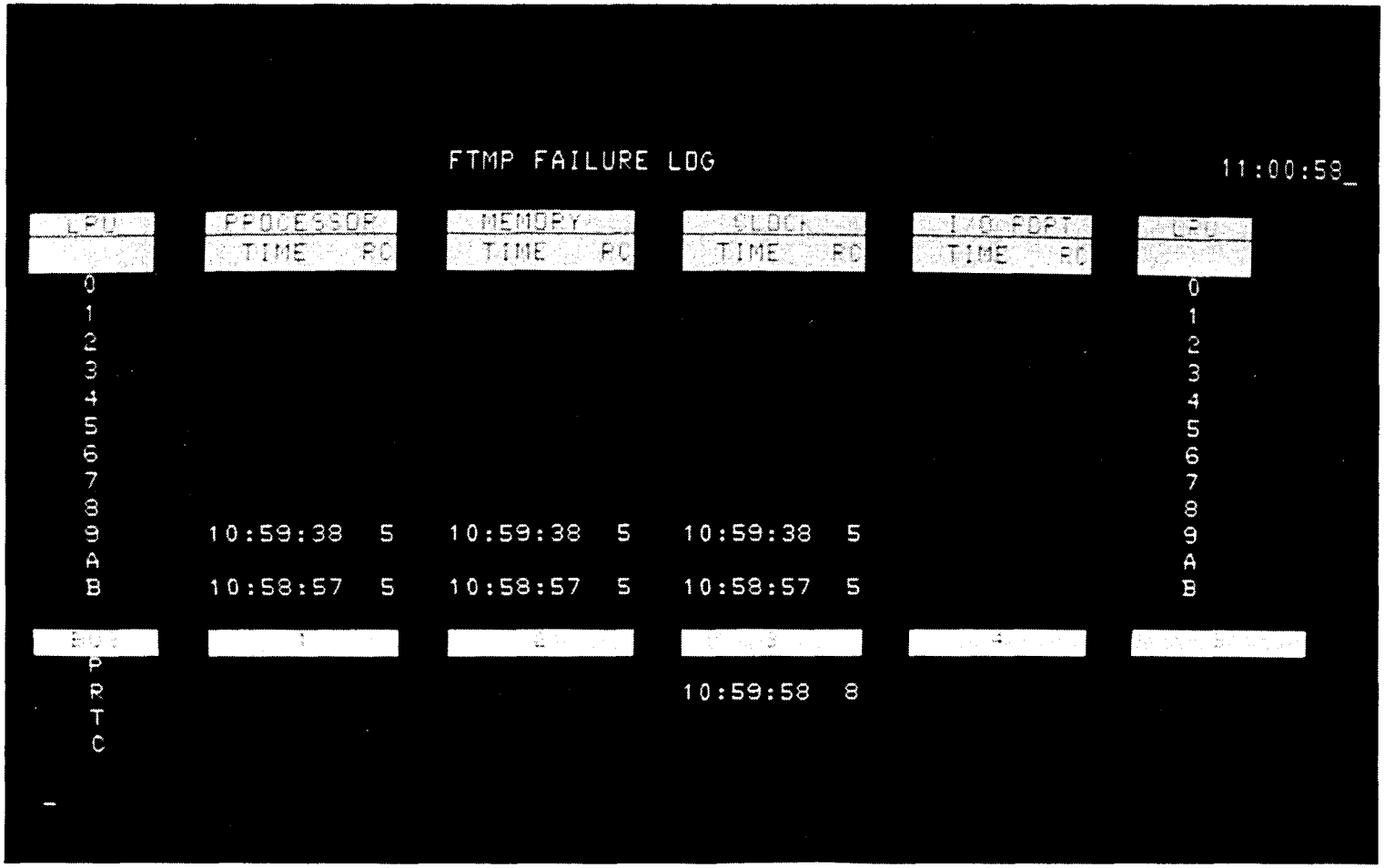


Figure 6-3. Failure Log Display.

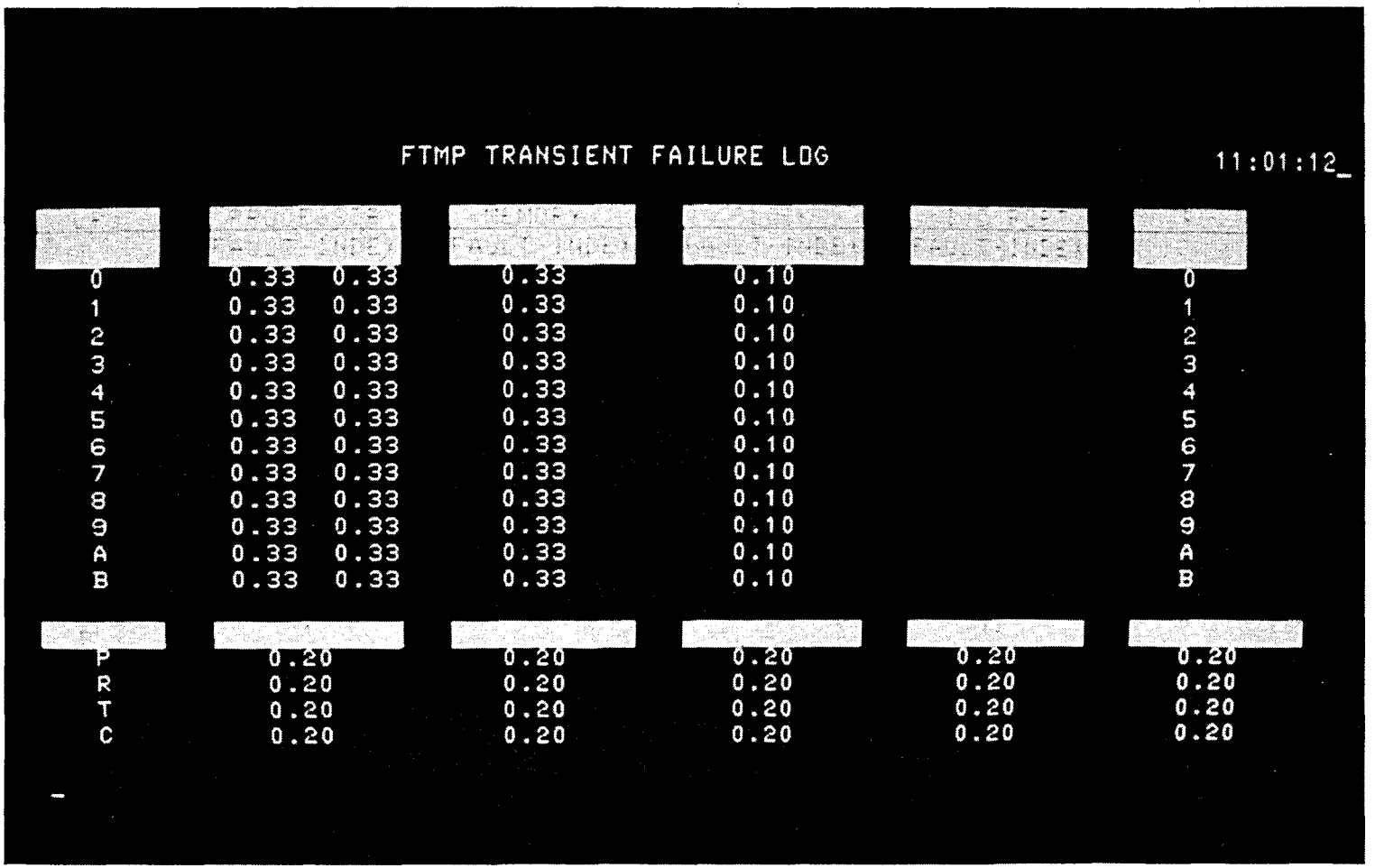


Figure 6-4, Transient Failure Log Display.

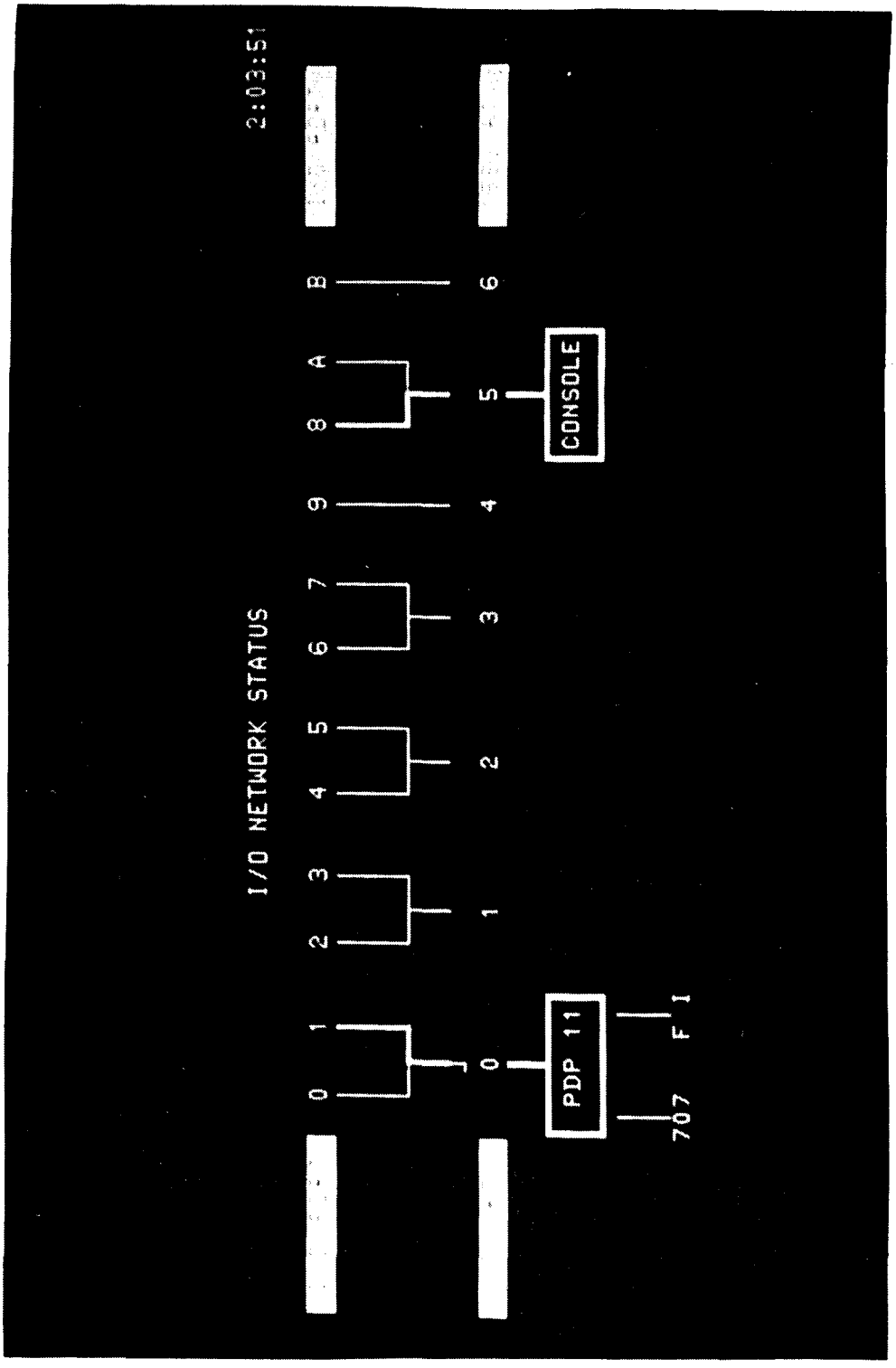


Figure 6-5. I/O Port Status Display.

output commands generated by the autopilot. This display is invoked by the command APFDS. It is shown in Figure 6.6.

The console commands accepted by the FTMP, in addition, to the four display commands just described, are as follows.

FAIL Pn This command is used to fail processor n. The display program in the FTMP does this by resetting the processor.

FAIL Mn Memory n can be failed by changing its MRR to 1E.

RESTORE P l,m,n... Processors l, m, n etc. may be repaired using this command. Their status in the system configuration tables is changed from 'failed' to 'spare'. Also, if the associated memory and/or clock in that LRU are marked failed they are restored to 'spare' condition. The fault-indices of these units are reset to their respective base values.

RESTORE m l,m,n... This command is used to repair several memories.

RESTORE c l,m,n... This command is used to repair several clocks.

RESTORE BUS P / R / T / C n This command is used to repair line n of P, R, T, or C bus.

CHANGE PORT N This command is used to change the I/O port configuration through which external devices are being accessed. PDP-11/60 interface ports can be changed from 0 to 1 and vice versa and display console ports can be changed from 8 to A and vice versa.

REFRESH Each display has two components, background, and foreground. The background is the part that does not change such as titles,

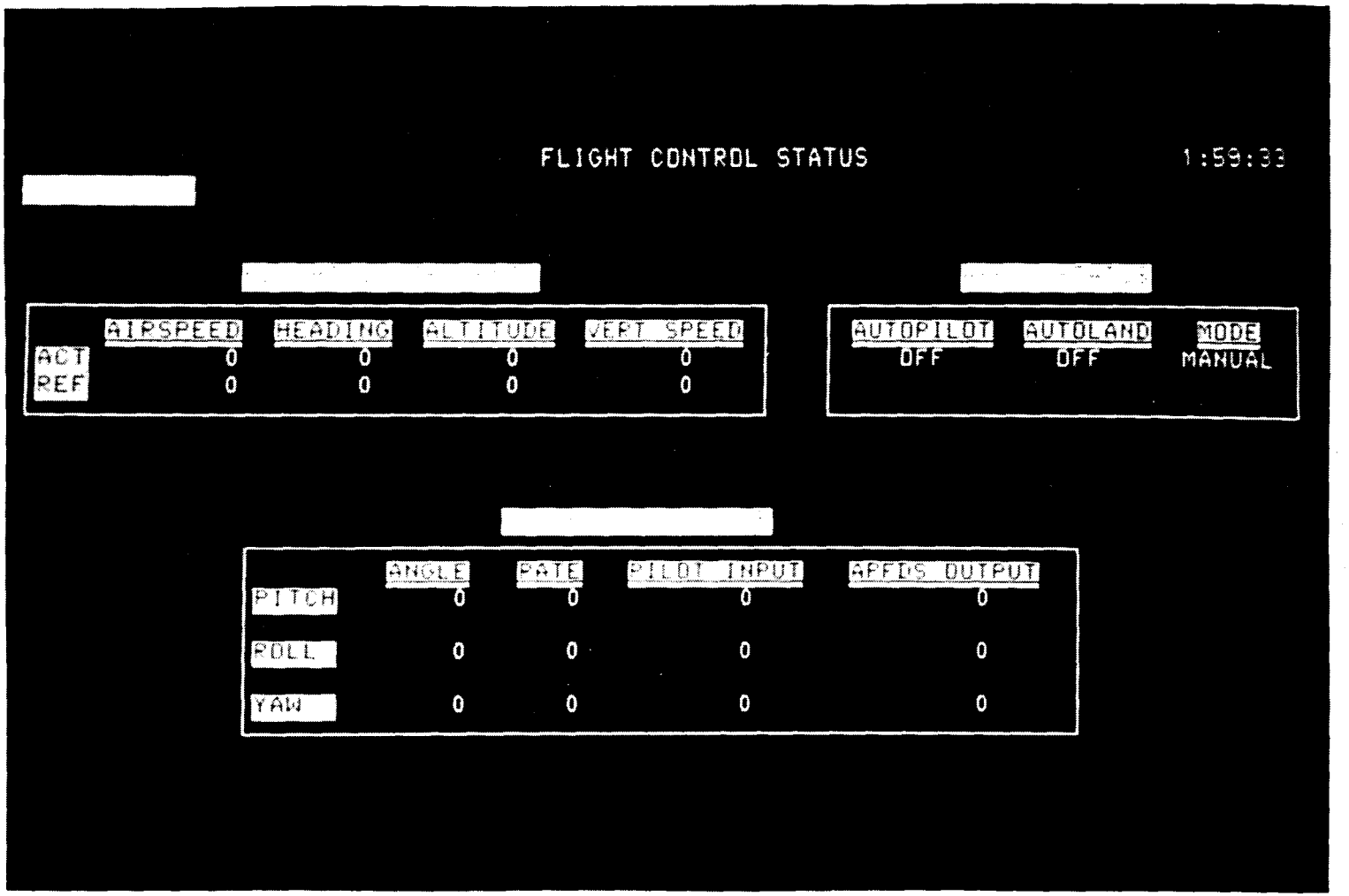


Figure 6-6. Autopilot/Flight Director Status Display.

background enhancements, etc. The foreground is the system configuration information. This command is used to update the foreground of the current display. Normally, only the time of day is refreshed constantly. Other foreground information is not changed unless there is a change in the system configuration.

TIME HR:MIN:SEC This command is used to set the time of day to HR, MIN and SEC. The display program sets a double word BASE.TIME in system memory such that the commanded time equals BASE.TIME + Real Time Clock. There is a TIME applications program that computes the current time by adding BASE.TIME to the Real Time Clock and stores this value in a double word TIME.NOW in the system memory. TIME.NOW represents the time of day based on a 24-hour clock in terms of quarter millisecond units. It has the same least count as the Real Time Clock. It is converted to hour, minute, second format by the display program and shown on the console.

RESTART System configuration tables are reinitialized and the system is restarted.

The display program runs as a privileged R1 rate group applications task. It has two major sections, input and output. The input section processes operator commands typed on the console. The output section formats display into ASCII character strings. Two circular buffers in the system memory are used to store the input and output characters. These are IN.BUFF and OUT.BUFF, respectively. IN.BUFF can hold 512 characters while OUT.BUFF can hold 2048 characters. The actual I/O transactions between the FTMP and the 1553 emulator are performed by

an I/O driver TTY which is detailed in Section 3.6. The display program retrieves characters from IN.BUFF and inserts characters in OUT.BUFF independent of TTY which simultaneously may fill IN.BUFF and empty OUT.BUFF. To accomplish this, two sets of pointers are used, IN.BUFF.IN and IN.BUFF.OUT for IN.BUFF and OUT.BUFF.IN and OUT.BUFF.OUT for OUT.BUFF.

To display a character on the screen, it is necessary that the cursor be in the required position, that is the column and row where the character is to be displayed. If it is not, it is moved to the correct row and column by an absolute or a relative cursor move command. Such commands are composed of several ASCII control characters. Control characters are also used to enhance display background. The background can be made half bright, inverse video, etc. Two character sets are available on the HP terminal, a standard alpha-numeric set and a line drawing set. Both are used in the FTMP displays. Field definition characters are used to switch between the two character sets.

The output section of the display program generates cursor positioning commands, field definition control characters and display characters and places these in the circular output buffer. The input section empties characters from the input buffer for the command processor and also places them in the output buffer with appropriate cursor positioning commands. This ensures that characters typed on the terminal are echoed back (and thus appear on the screen) on the bottom line which is reserved for operator commands. Pointers in the system memory are

used to store the actual cursor position on the screen and the next echo column. The echoing function is handled by a procedure called IOECHO.

To relieve the main display program, which incidentally is in the library FTMP.AED(DISPLAY), from detailed cursor positioning, ASCII character conversion and buffer management tasks a set of basic output primitives has been written. These primitives are in the library FTMP.AED(OUTPUT). These routines position cursor as required, format ASCII character strings, place characters in the output buffer and update buffer pointers. Following is a brief description of these routines.

CR, LF, CRLF These routines perform the functions of carriage return, line feed, and carriage return-line feed, respectively. CR moves the cursor to column 0 if no 'virtual screen' is defined. However, a virtual screen may be defined by using routines ROW and MARGIN. In that case, the cursor is moved to the column where the virtual screen begins.

CLEAR SCREEN This routine clears the screen and moves the cursor to column 0 and row 0, that is, the upper left hand corner of the screen.

HOME This routine moves the cursor to the upper left hand corner of the real screen or virtual screen if one is defined.

ROW(M) This routine defines the top of the virtual screen to be M.

MARGIN(N) This routine defines the left hand margin of the screen to be N.

INCR.COL(M) This routine moves the cursor by M columns. M may be positive or negative.

INCR.ROW(N) This routine moves the cursor by N rows. N may be positive or negative.

OUT.HEX.CHAR(I) This routine converts hexadecimal number in the least significant 4 bits of I into its ASCII representation and displays this character at the present cursor position.

OUT.2HEX.CHAR(I) This routine is similar to the previous one. It displays two hexadecimal numbers contained in the least significant byte of I.

OUT.4HEX.CHAR(I) This routine displays four hexadecimal numbers contained in I.

OUT.CHAR(I) This routine displays the ASCII character contained in the lower byte of I.

OUT.2CHAR(I) This routine displays 2 characters contained in I.

OUT.CNTL.CHAR(I) This routine transmits the control character in the low byte of I to the terminal. Since this is a control character the cursor position is not changed.

OUT.2CNTL.CHAR(I) This routine transmits 2 control characters to the terminal.

OUT.MSG(P) This routine displays a string of ASCII characters pointed to by P.

There are other routines that process the input from the console. These are as follows.

GET.LINE This routine checks to see if there is any input from the terminal. Terminal input is not processed until the input line is

terminated by a carriage return. If such an input line exists, this procedure obtains it from appropriate buffers in the system memory.

HEX.INPUT(B) This integer routine scans the input line for hexadecimal number. It returns the number, if found, as the procedure value and the argument B is set to true. If the next character in the input stream is not a valid hexadecimal number, B is set to false.

COMPARE(P) This boolean procedure compares the input stream with a character string pointed to by P. The procedure value is set to true if a match is found.

SKIP.FIELD(B) This routine scans the input stream and positions a cursor or pointer at the next character after skipping a field. Fields are separated by spaces, periods, commas or colons. B is set to true if end of line is reached.

6.2 Flight Control Software

The flight control system programmed on the FTMP is a totally automatic flight control system (AFCS). It is patterned after the Lockheed L-1011 Tristar avionic flight control system. It provides automatic aircraft control from take-off to touch-down. The flight control software can be divided into two major parts, autopilot flight director system (APFDS) and autoland. These are described in the following two subsections.

6.2.1 APFDS

The autopilot/flight director can be used by the pilot to provide him assistance in varying degrees. This ranges from basic stability augmentation to completely automatic flight control. The stability augmentation system (SAS) provides artificial pitch and roll damping when the autopilot is engaged. In addition, yaw damping is provided at all times. The yaw SAS also commands rudder to coordinate turns and provides runway alignment and rollout during autoland operations.

The autopilot may be engaged in one of several modes. If the pilot wishes to maneuver the aircraft with the help of the autopilot, it may be engaged in attitude hold/control wheel steering (CWS) mode. In this mode, the autopilot holds a constant pitch and roll attitude and the pilot's pitch and roll stick inputs are interpreted as rate commands to change the aircraft attitude.

For the climb and descent phases of flight, two different modes of operation are available. These are the vertical speed hold and Mach hold. Altitude hold is available for the cruise flight phase. During cruise, altitude or Mach hold may be engaged.

For directional guidance and control, a heading hold mode is also available. Heading hold may be engaged in conjunction with any of the other autopilot modes. The implementation details of various APFDS modes are described below.

CWS Figures 6-7 and 6-8 show the pitch and roll loops of the CWS mode, respectively.

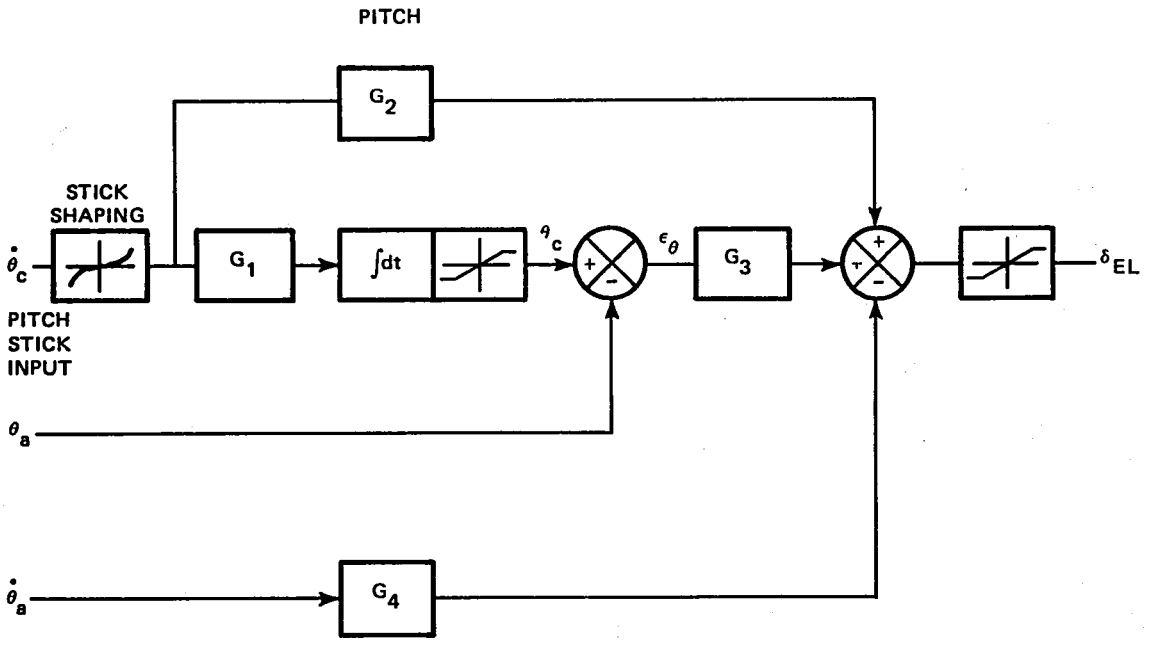


Figure 6-7. CWS - Pitch loop.

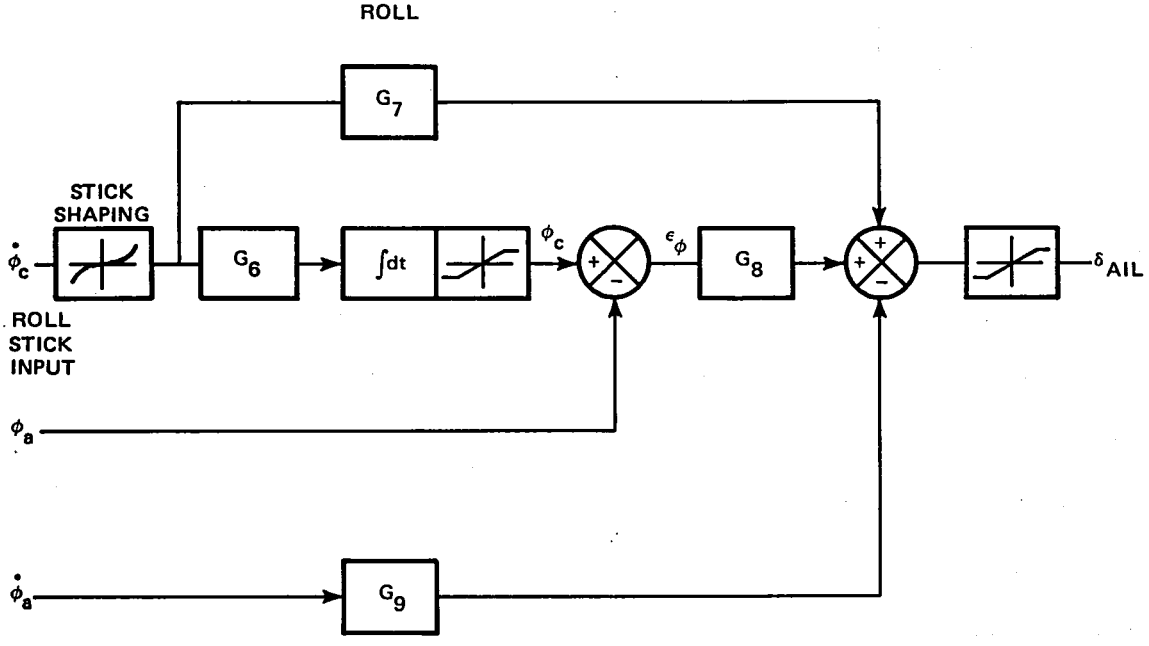


Figure 6-8. CWS - Roll loop.

In the pitch loop, the pilot's pitch stick input is interpreted as pitch rate command. This input is passed through a stick shaping filter to provide a dead zone around the neutral stick position and it is squared thereafter to provide the pilot rapid response. This reshaped stick command is integrated to obtain the desired pitch angle. This is compared to the actual pitch angle of the aircraft to generate an error signal. The error signal is amplified through a gain to provide the basic elevator command. This command is modified slightly for two reasons. First, a negative feedback through pitch rate is used to provide damping. Second, the pilot's stick input is added to the elevator command for rapid response.

The roll loop is identical to the pitch loop. Instead of pitch angle and rate, roll angle and rate are used to provide an aileron command.

In order to provide a smooth transition from manual flying mode (autopilot not engaged) to autopilot mode, the pitch and roll loops run open loop when the autopilot is not engaged. This ensures that the control surface positions will not change abruptly at the instant the autopilot is engaged. The CWS program runs as a privileged R4 applications task. It obtains the required sensor values from a sensor buffer area in the system memory and places the actuator commands in an actuator buffer area, also in the system memory. The actual sensor/effector I/O is performed by an I/O driver in the R4 dispatcher prolog. This is explained in Section 3.6.

LATERAL SAS Figure 6-9 shows a block diagram of the lateral stability augmentation system. This program runs as an R3 rate group privileged task. The pilot's yaw input (rudder pedals) is passed through stick shaping and compared to the side slip angle to generate an error signal. A wash-out based on yaw rate is added to the error signal. The final error signal is amplified to produce the rudder command. This program is run all the time whether the autopilot is engaged or not. It provides basic stability augmentation and turn coordination.

ALTITUDE HOLD Figure 6-10 shows a block diagram of the altitude hold mode of the autopilot. This program runs as an R1 rate group privileged task. An elevator command is produced in response to an error in the altitude. Rate of change of altitude and the pitch rate are also fed back to provide smooth altitude capture and damping. A constant altitude is maintained by varying the aircraft pitch attitude which in turn varies the airspeed since the throttle position is not moved.

MACH HOLD In this mode of the autopilot, shown in Figure 6-11, a constant Mach number is maintained by changing the aircraft pitch attitude which in turn results in a change in altitude since the throttle position is not moved. An error signal generated by a deviation from the reference Mach number is amplified and fed back on the elevator command. Rate of change of airspeed and pitch rate are also fed back for smooth Mach capture and damping. This task runs at R1 frequency.

VERTICAL SPEED HOLD In this mode, shown in Figure 6-12, the vertical speed is held constant by controlling the aircraft pitch attitude. An error signal produced by a deviation from the reference verti-

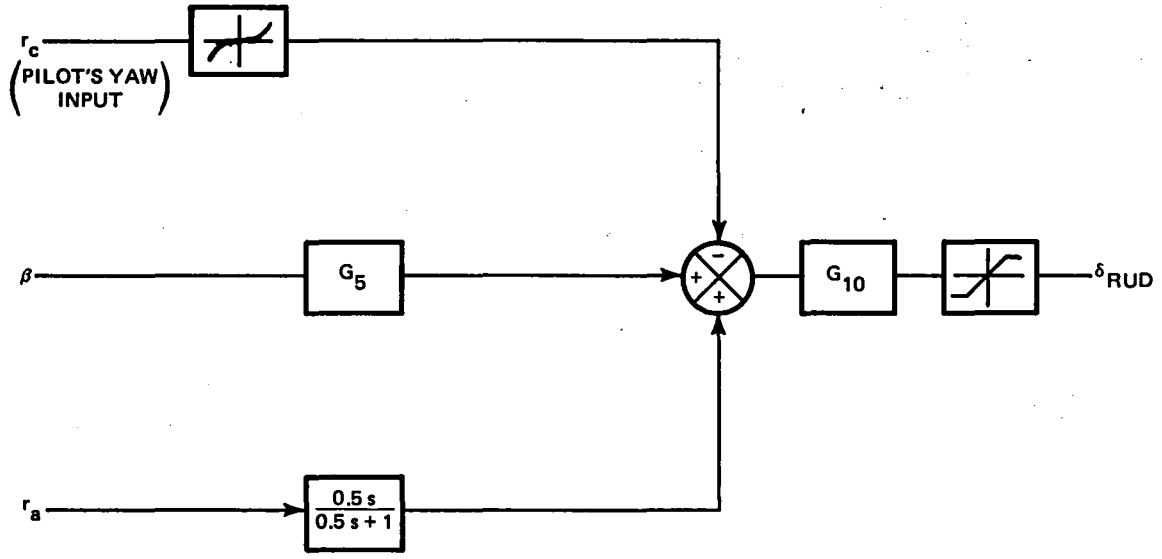


Figure 6-9. Lateral SAS.

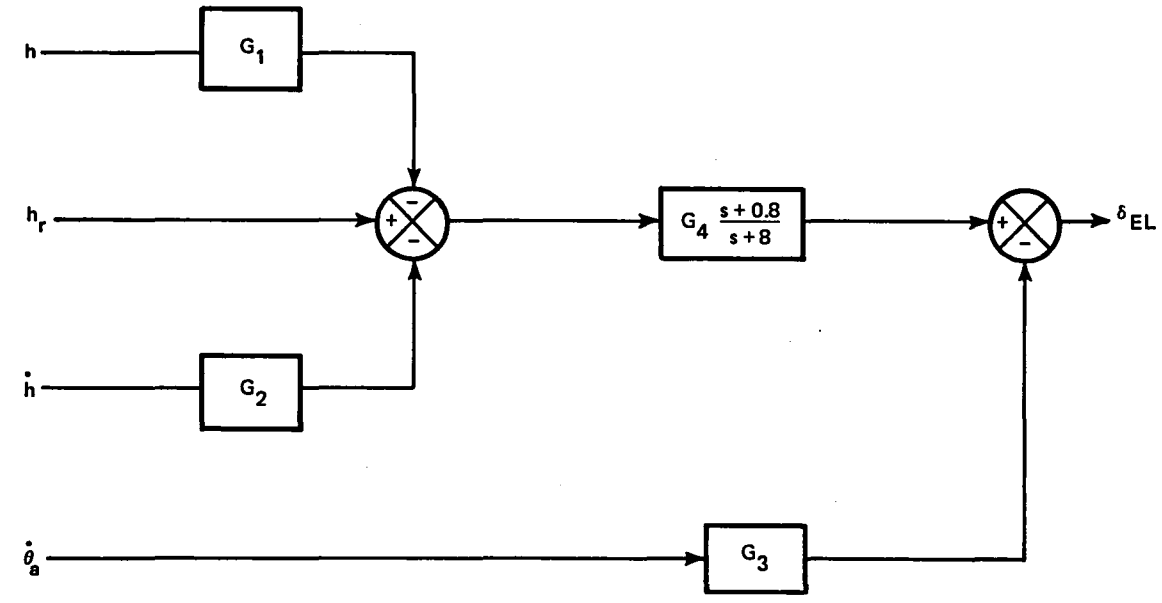


Figure 6-10. Altitude hold.

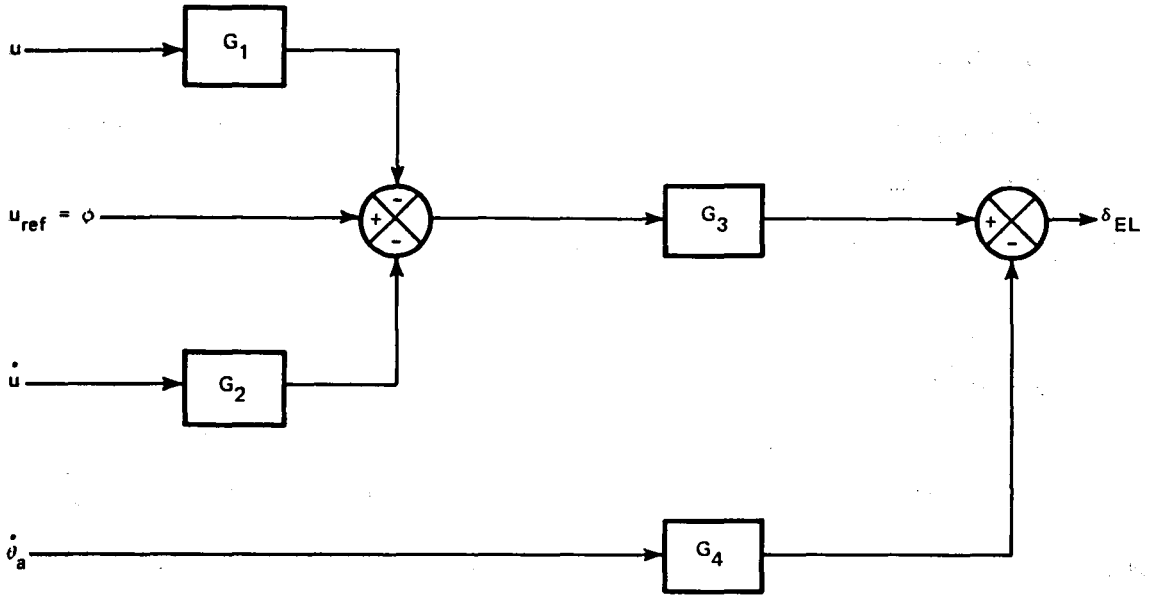


Figure 6-11. Mach hold.

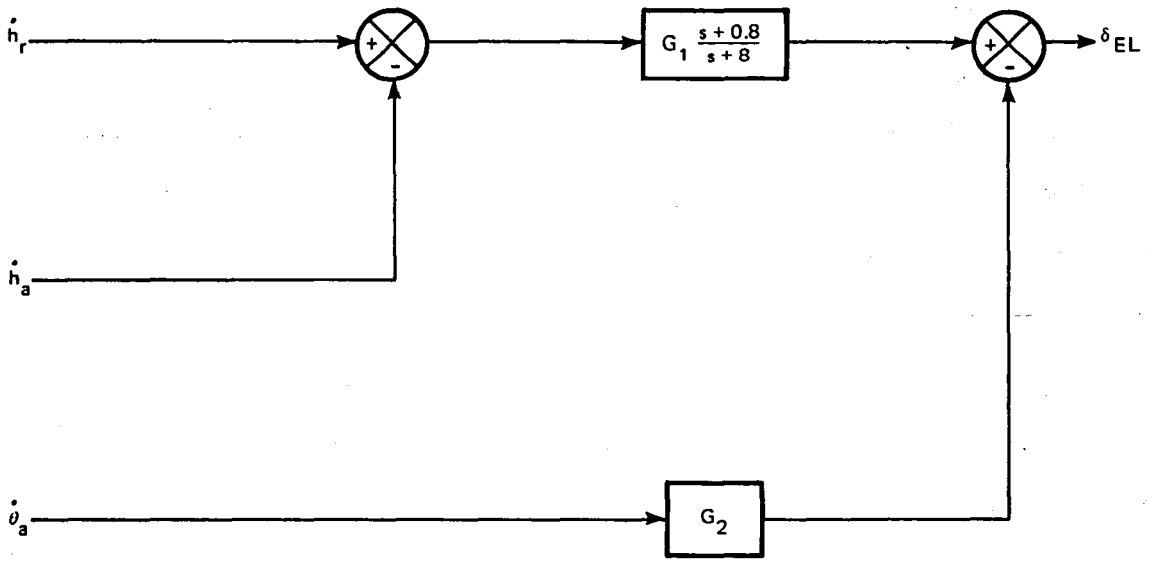


Figure 6-12. Vertical speed hold.

cal speed is used to command the elevator. Pitch rate feedback is used for damping. The vertical speed is held constant at the expense of the airspeed. This task runs at R1 frequency.

HEADING HOLD This mode is shown in Figure 6-13. A deviation from the reference heading is used to produce a bank angle which is fed as input to the CWS roll loop. This task runs at R1 frequency.

6.2.2 Autoland

Autoland may be used to do a completely automatic landing, including approach, flare, touch-down, and roll-out. The approach is made by capturing the ILS (Instrument Landing System) signals. Localizer signal is tracked for lateral guidance while the glideslope signal is followed to maintain the correct glide path. A constant airspeed may optionally be maintained if the autothrottle mode of the autoland is engaged. A brief description of each of these modes follows.

AUTOTHROTTLE Figure 6-14 shows a block diagram of this mode. A deviation from the reference airspeed produces an error signal that is amplified and converted into a throttle command. Acceleration feedback is used for damping. This task, as all other autoland tasks, runs as a privileged R4 rate group applications program.

GLIDESLOPE Figure 6-15 shows the transfer function used to produce the glide slope. The glide slope deviation signal is used to produce a pitch angle command which is fed to the CWS pitch loop.

FLARE This mode, shown in Figure 6-16, is automatically engaged at a radio altitude of 15 feet. The glide slope mode is disengaged at

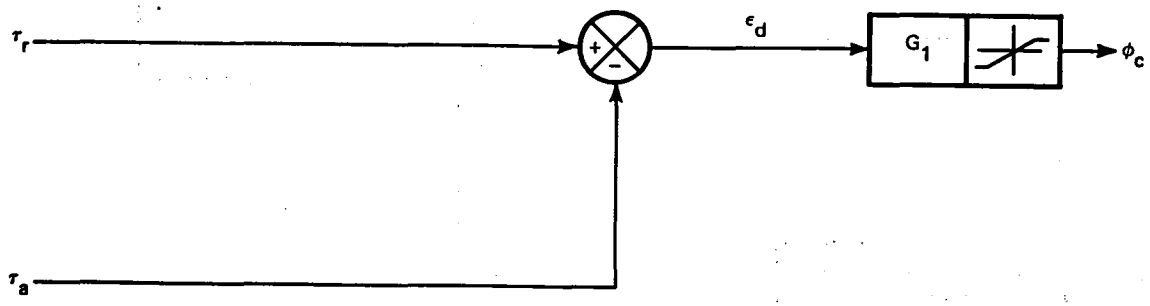


Figure 6-13. Heading hold.

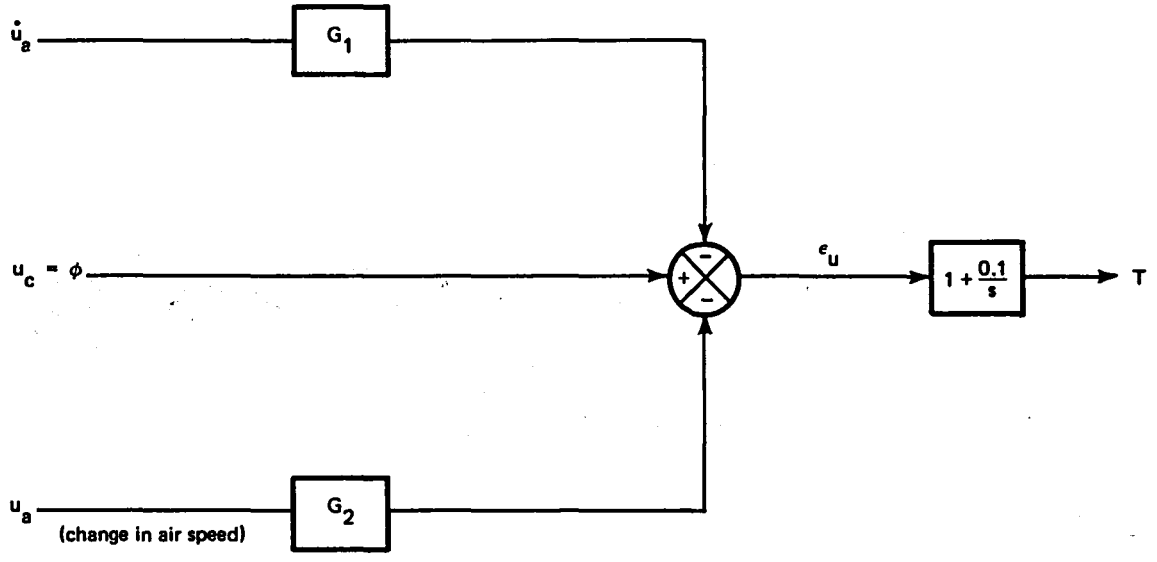


Figure 6-14. Autoland - Airspeed.

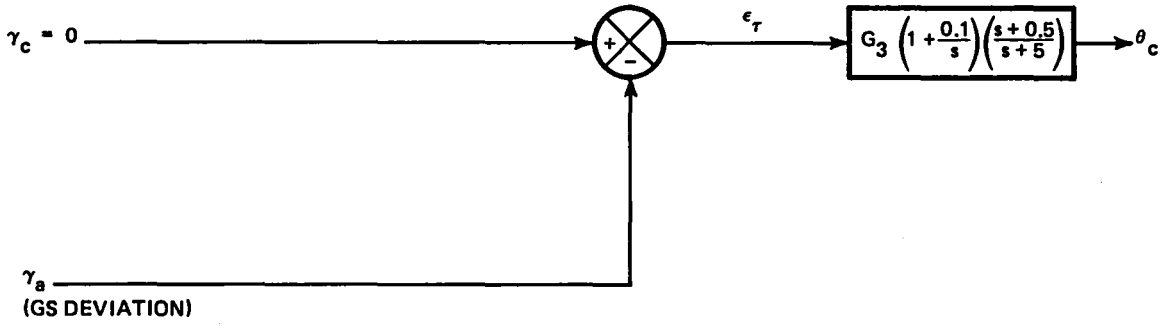


Figure 6-15. Autoland-Glide slope.

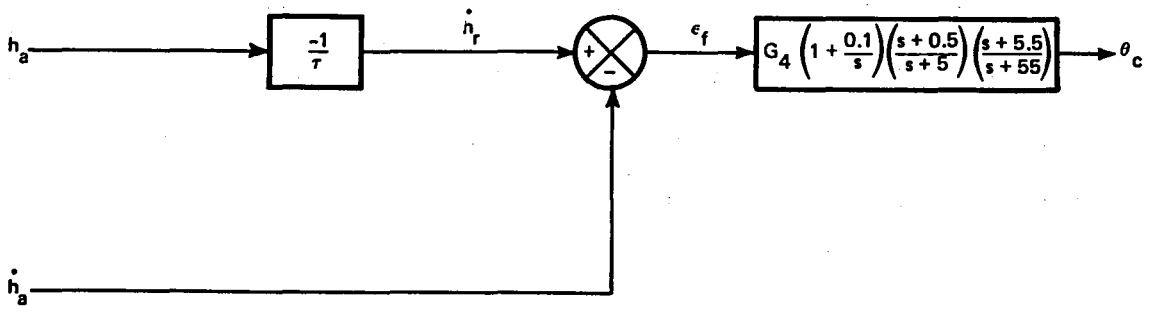


Figure 6-16. Autoland-flare.

this point. The attitude above the runway and the rate of descent are used to produce a pitch angle command which is fed as input to the CWS pitch loop. The object here is to reduce the rate of descent to zero just when altitude reaches zero.

LOCALIZER The lateral autoland mode is shown in Figure 6-17. This is used to track the runway centerline by following the localizer deviation signal. A yaw rate is produced as the output of this loop which is fed to the lateral SAS loop. This same mode is used for VOR tracking although the gains are slightly different in the VOR mode.

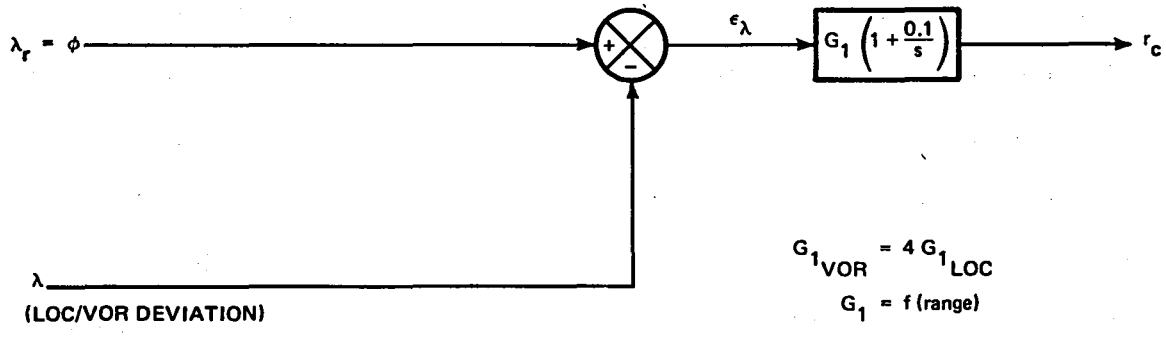


Figure 6-17. Autoland - Lateral.

6.3 FTMP Core Software Summary

Figure 6-18 shows a memory map of the FTMP load module produced by the CAPS link editor. The load module consists of all of the Executive, Application, and Self-Test software described in the preceding chapters.

Each program listed in the memory map is compiled and assembled independently to produce a relocatable object code. All of these object codes are then linked together by the link editor to produce an absolute load module. For each program the memory map shows three pairs of hexadecimal addresses under the headings of Counter 0, Counter 1, and Counter 2. Counter 1 shows the virtual starting and ending address of the executable code. Counter 0 shows the real starting and ending address of the static read/write area (work area) for the program, and Counter 2 shows the real starting and ending addresses of the read only data (constants used by the program).

For example, the code for the R4 task dispatcher 'DISPR4' occupies the virtual address space BF4 to E75. It uses three constants which reside at B0 to B2, and it uses two statically allocated variables that reside at 2407 and 2408. A procedure may also use a dynamically allocated work area. This area is temporarily allocated on the stack when the procedure is invoked and released when the procedure retires.

The virtual address space for the code area is unity mapped into the 8K processor PROM for the address space 0 to 1FFF. Most of the frequently used Executive routines occupy this address space. That is, these routines are permanently loaded into the PROM of each processor. Referencing Figure 6-18 once again, these routines are listed under the

Module	Counter 0	Counter 1	Counter 2	Comments
		SEGMENT PROM		Cache PROM Segment
REGS				Hardware register definitions
KERNEL	2000 23D7	0100 01FA	0000 009F	Context switch routine
DOIT		01FB 0212		Implements AED 'DOIT' Facility
CHFA		0213 021E		
NTIMERS	23D8 23E8	021F 0448	00A0 00A0	Interval & Real-Time Clock Routines
CACHE	23E9 23EE			Cache Memory Globals
PEND		0449 045B		Process Pend Routine
SERVICE		045C 054C		System Memory, and I/O Access Routines
RESTART		054D 076A	00A1 00AA	System Bootstrap Routine
GPROCS		076B 0898	00AB 00AC	General Procedures (Lock, Unlock etc.)
READEL		0899 08B8		
READPRTC		08B9 08F8		Error Latch Read & Clear Routines
CLRALLEL		08F9 0917	00AD 00AD	
TA		0918 0984		Test Adapter Interface
PFAULT	23EF 2406	0985 09FF		Page Fault Handler
NDISPR1		0A00 0B07	00AE 00AE	
NDISPR3		0B08 0BF3	00AF 00AF	R1, R3 & R4 Task Dispatchers
DISPR4	2407 2408	0BF4 0E75	00B0 00B2	
IDLE		0E76 0E96		Idle Process
OUTPUT	2409 2484	0E97 1474	00B3 00C4	
TTY		1475 1626	00C5 00C9	Input/Output Handlers
NIO1553		1627 173D	00CA 00CD	
RECONF		173E 1961	00CE 00CF	Configuration Command Generator
SWAPCMND		1962 1E23	00D0 00D5	Spares Cycling Command Generator
TIME		1E24 1E47	00D6 00D6	Time of Day
		SEGMENT M000		System Memory Segment
SCC	2500 257A	2800 3C35	3C36 3C56	System Configuration Controller
SELFTEST		3C57 3D1B	3D1C 3D21	
RCVTEST		3D22 3E45		
XMITTEST		3E46 3F59		
POLLTEST		3F5A 4018		System Self-Test Programs
CLKTEST		4019 40B4		
CHECKSUM		40B5 40E5		
VOTER		40E6 40F5		
DISPLAY		40F6 5562	5563 57D1	
READALL		57D2 580D		Console Display Routine
CONCOM		580E 64F4	64F5 64F6	Configuration Command Processor
SWAPLRUS		64F7 686F	6870 6872	Spares Cycling Command Processor
B707		6873 6968		Boeing 707 Cockpit Display Interface
NAPFDS	2700 271C	6969 6CD6	6CD7 6D26	Autopilot/Flight Director
FLOAT		6D27 6E30		Floating Point Arithmetic Routines
TASKR43	2700 2700	6E31 6E39		
TASKR32	2600 2600	6E3A 6E42		Dummy Applications Tasks
TASKR33	2600 2600	6E43 6E4B		
TABLES	0000 0EA3			Shared Data Resident in System Memory

Figure 6-18. FTMP Load Module Memory Map.

segment PROM. Examples of some of these Executive procedures include the context switch routine (KERNEL), system bootstrap routine (RESTART), routines to access system memory, I/O registers, real-time clock (SERVICE), task dispatchers for the three rate groups (NDISPR1, NDISPR3, DISPR4), routines to read error latches (READEL, READPRTC, CLRALLEL), input/output handlers (OUTPUT, TTY, NIO1553) and procedures to cycle spares (SWAPCMD) and carry out system reconfiguration commands (RECONF).

The next 2K of virtual address space (2000-27FF) is unity mapped into the processor RAM address space 2000-27FF. This area is used for stacks, Process State Descriptors (PSDs), and system globals. Figure 6-19 shows the relationship between various FTMP address spaces.

The rest of the FTMP load module is located in the virtual address space 2800-6E4B. It is loaded in the same physical addresses of the shared memory as well. Most of the system configuration controller, self-test programs, autopilot and autoland applications software, and the shared data is loaded in the system memory. Since processors can execute code only from their respective cache memories, programs located in the system memory must first be read into the cache RAM before they can be executed. A mapper table in each processor maps virtual address space to physical address space and also indicates whether a given page (256 words) is loaded in the physical memory, that is, the cache RAM. If the required instruction is not present in the cache memory, a page fault interrupt is generated. The page fault handler then reads in the needed page from the system memory into the cache memory.

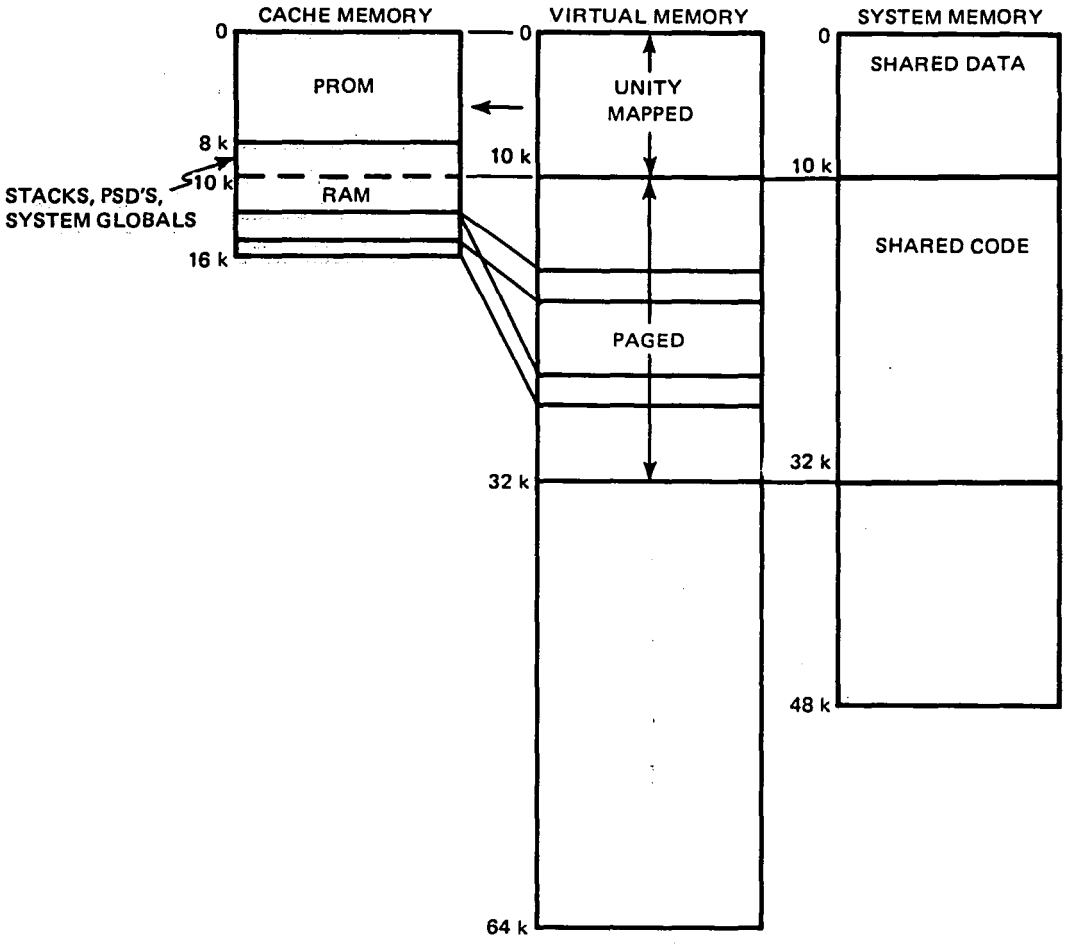


Figure 6-19. FTMP address spaces

The load module is slightly different when the faults are being inserted into the FTMP. The normal system configuration control program, SCC, is replaced by FSCC, which is designed to handle the special fault insertion set-up.

CHAPTER 7

SUPPORT SOFTWARE

The FTMP software is written either in a high-level language called AED or in the CAPS assembly-level language. There are no facilities on the FTMP itself to compile or assemble these programs. The source programs must be converted into the CAPS machine language on one of the mainframe computers such as IBM 360/370, UNIVAC 1108, or Amdahl 470. At CSDL the required support software has been installed on an Amdahl 470/V8.

There is an AED cross-compiler that is used to compile and assemble AED source programs to produce relocatable object code modules. The AED source programs reside in a partitioned data set PDP1160.FTMP.AED. The object code modules are placed by the compiler in a partitioned library PDP1160.FTMP.COBJ. In addition, the compiled output is printed and an archival copy placed in a partitioned list file PDP1160.FTMP.LIST.

There is a CAPS cross-assembler that assembles source programs written in the CAPS assembly language and produces a relocatable object code module. The CAPS source programs reside in the partitioned data set PDP1160.FTMP.ASM. The object code and list files are stored in the same libraries used by the cross-compiler.

Various relocatable object modules can be linked together using the cross-linker to produce an absolute load module. The load modules are stored on the Amdahl in a library PDP1160.FTMP.LOAD. Link files are stored in PDP1160.FTMP.LINK while the list output of the linker is stored in PDP1160.FTMP.LINKLIST.

There is also a host compiler for AED that compiles AED source programs into the 470 executable machine language. The host compiler is mainly used to compile the AED cross-compiler, cross-assembler and cross-linker all of which are written in AED.

The standard collins linkage editor has been modified for the FTMP use. The modified linker accepts three different directives to produce three different types of records in the load module. These are as follows.

SEGMENT PROM This directive is used to direct the linker output to the FTMP PROM memory. The linker prefaces all records produced after this segment with the letter P. Since the PROM address space is 0 to 1FFF, the linker flags any code that is outside this address space. The PROM programmer software package extracts only the P records of the FTMP load modules.

SEGMENT RAM The linker prefaces records produced after this directive with R and checks the addresses for the range 2000 to 3FFF.

SEGMENT Mnnn This directive is used to direct the linker output to the FTMP system memory. The linker prefaces these records with the letter S. The loader on the PDP-11 examines the first letter of each record and loads it in the cache or the system memory depending upon the

type of record. Code nnn in the directive is used to relocate the load module to a system memory address that is different from the virtual memory address space in which the load module is to be run. If nnn is zero, the system memory address and the virtual memory address are the same. If it is not zero, the object code is moved from its virtual address space by nnn pages. This may be a positive or a negative offset. Figure 6-19 shows the relationship between various FTMP address spaces.

The commands to invoke the cross-compiler, assembler and linker on the TSO are contained in the partitioned library 'PDP1160.FTMP.CMD.CLIST' and are as follows.

CAEDC name This command invokes the AED cross-compiler. The AED source program 'userid.FTMP.AED(name)' is compiled on-line and a listing is produced in the file 'userid.FTMP.LIST(name)'. Compilation errors are also displayed on the terminal. If the user has changed his prefix to ID other than his own, the source and list files are accessed from the prefix ID library.

CAEDCA name This command is similar to CAEDC. It invokes the AED cross-compiler. If the compilation does not produce an error code of severity higher than 2, the cross-assembler is invoked. The assembler produces a relocatable object code file 'Prefix id.FTMP.COBJ(name)'.

CASM name This command invokes the cross-assembler. The CAPS assembly language source program 'prefix id.FTMP.ASM(name)' is assembled on-line. The assembler produces a relocatable object module 'prefix id.FTMP.COBJ(name)' and a list file 'prefix id.FTMP.LIST(name)'.

CLINK name This command invokes the cross-linker. The linker directives are obtained from the data set 'prefix id.FTMP.LINK(name)' and the object code libraries referenced in the directives are obtained from 'prefix id.FTMP.COBJ'. The linker produces an absolute load module 'prefix id.FTMP.LOAD(name)' and a list file 'prefix id.FTMP.LINKLIST(name)'.

All of the above commands may be used to submit batch jobs on the Amdahl by using the keyword BATCH.

REFERENCE

1. Nassi, I., and B. Shneiderman, "Flowchart Techniques for Structured Programming," SIGPLAN Notices, August 1973.

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-166072		

4. Title and Subtitle	5. Report Date
DEVELOPMENT AND EVALUATION OF A FAULT-TOLERANT MULTIPROCESSOR (FTMP) COMPUTER Volume II - FTMP Software	May 1983
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
J. H. Lala and T. B. Smith III	CSDL-R-1601
	10. Work Unit No.
9. Performing Organization Name and Address	
The Charles Stark Draper Laboratory, Inc. 555 Technology Square Cambridge, Massachusetts 02139	11. Contract or Grant No. NAS1-15336
	13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address	
National Aeronautics and Space Administration Washington, DC 20546	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Minitor: Charles W. Meissner, Jr.
Final Report

16. Abstract

This report is Volume II of a four-volume report on the Fault-Tolerant Multiprocessor (FTMP) project. It covers in detail the software developed for the FTMP.

The FTMP Executive is a timer-interrupt driven dispatcher that schedules iterative tasks which run at 3.125, 12.5, and 25 Hz. Major tasks which run under the Executive include System Configuration Control, Flight Control, and Display. The Flight Control task includes Autopilot and Autoland functions for a jet transport aircraft. System Displays include status displays of all hardware elements (processors, memories, I/O ports, buses), failure log displays showing transient and hard faults, and an Autopilot display.

All software is in a higher order language (AED an ALGOL derivative). The executive is a fully distributed general purpose executive which automatically balances the load among available processor triads. Provisions for graceful performance degradation under processing overload are an integral part of the scheduling algorithms.

17. Key Words (Suggested by Author(s))		18. Distribution Statement
Interaction Consistency	Synchronous	FEDD-Distribution
Multiprocessor	Fault-Tolerant	
Distributed Executive		
Reconfigurable		
Interrupt Driven Executive		Subject Category 62

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	234	

Available: NASA's Industrial Applications Centers

End of Document